An Investigation into Cooperative Behaviour: Altruism and Evolutionary Computing

Tim Watson

Submitted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy

De Montfort University

April 2003

Abstract

Tim Watson **2003**

An Investigation into Cooperative Behaviour: Altruism and Evolutionary Computing

This thesis describes how individuals in an evolving population can be encouraged to cooperate. The ability to evolve a cooperating population is of obvious benefit to many living things but, to date, no truly cooperative evolutionary computing systems have been produced, in spite of the fact that such systems would be of enormous benefit.

After an introduction to the research topic, the thesis reviews the relevant literature and examines the various mechanisms through which cooperation is said to occur. The first significant research contribution of the thesis is a demonstration that the standard explanation for the evolution of cooperation – kin selection – only produces a briefly cooperative population, which soon reverts to non-cooperation. The second significant contribution shows that cooperation can fail to develop because an evolving system's environmental rate of change can outpace the speed of adaptation of its population, and identifies two techniques to counter this effect, leading to a cooperative system that can cope more easily with fast-paced environments. The final contribution is a mechanism that overcomes the inherent instability of kin selection, allowing systems to evolve cooperation.

While the work presented here concentrates on evolutionary computing systems based on genetic algorithms, some of its findings appear to have wider applicability.

Contents

List of Tables

List of Figures

Acknowledgements

The author would like to thank both Peter Messer and Peter Innocent for their continued supervision of this work. Particular thanks go to Peter Messer for the immense amount of time and patience he has shown when faced with countless overly long, detailed and muddled explanations of a variety of obscure ideas. It is to his credit that some of them now have merit.

Chapter 1

Introduction

All purposeful behaviour, unless it is centrally controlled, depends upon co-operation. From the function of a single cell to the strategic actions of an ambitious politician, if it is complex enough to have both purpose and behaviour then its constituent parts, or its environment, will have to cooperate.

In biological systems cooperation often emerges spontaneously: the harmony between DNA, RNA and protein, and between the nucleus, ribosomes and mitochondria within a single human cell; the cooperative societies of ants and bees; the caring of parents for their young; the mutualistic relationship[1] between a fungus and a green alga to form lichen. All of these forms of cooperation have evolved naturally. In many artificial environments a similar evolution occurs: cartels form to protect business profits from competitive pricing; opposition MPs will pair up to avoid unnecessary voting; during the first World war opposing soldiers would often deliberately avoid

[1] Often described less accurately as a symbiotic relationship. Lawrence (1995) defines symbiosis as a 'close and usually obligatory association of two organisms of different species living together, not necessarily to their mutual benefit; often used exclusively for an association in which both partners benefit, which is more properly called mutualism.'

injuring each other (Axelrod 1984, pp. 60–61). In spite of this, in almost all computing environments, if cooperation is needed it has to be engineered in. The development of sophisticated protocols and algorithms for scheduling, routing, parallel processing and networking takes years of intense effort and large amounts of money; yet in many environments it would seem that cooperation emerges for free.

It is easy to see why some simple forms of cooperation have developed. In England, we drive on the left. It is said that this is because in earlier and more lawless times when a rider passed another on the road it was in his interest to have his sword arm between himself and the other rider. Everyone cooperated and rode on the left because it was not only best for the population as a whole, it was also in the selfish interests of each rider. The cooperation still persists today: any driver who tries non-cooperation is likely to find his ability to pass his genes on to the next generation dramatically reduced. This is trivial cooperation because there is no conflict between the best choice for the individual and the best for the group.

Centralised control is another way to produce cooperative populations: the French drive on the right because Napoleon decreed it (Napoleon was left-handed). Yet, if centralised control is used to produce anything other than the simplest cooperation then the design of the central controller is often extremely difficult, if not virtually impossible.

Without centralised control and when the best for the individual conflicts with the best for the group, it is more difficult to see how cooperation can emerge and how it is maintained. Two main theories are employed to explain the phenomenon: kin selection and reciprocal altruism. Kin selection

requires that individuals recognise their close relatives. Reciprocal altruism requires that individuals repeatedly interact during their lifetime and learn to cooperate. However, it is not always possible, or desirable, to allow repeated interaction within a single generation (for reasons of computational efficiency or because an interaction can result in the death of an individual). Consequently, a computer system that encourages cooperation, even when individuals only interact once, should be based on kin selection.

While previous work in fields such as artificial life, distributed artificial intelligence and machine learning has produced limited success with cooperative systems (Frean & Abraham 2001, Ackley & Littman 1994, Axelrod 1984, Ashiru & Czarnecki 1997), the understanding necessary to allow computing agents to cooperate is still largely lacking. Consequently, any progress gained from research into cooperation should add valuable knowledge to a poorly understood area, which has the potential to provide enormous benefits to many areas of computer science.

1.1 Identifying a Research Topic

The initial choice of research topic: 'An Investigation into Cooperative Behaviour' was decided upon after consideration of a number of research topics related to artificial intelligence, an area in which the author is particularly interested. The choice was eventually made to study cooperative behaviour as it was felt that the large body of associated research literature had resulted in limited progress for cooperative computing environments. Many areas of computer science research that would benefit from a cooperative environment, such as scheduling, routing and parallel problem solving, were

still struggling to take advantage of cooperation. Consequently, any progress in the topic would have a reasonable chance of providing useful advances in a number of application areas.

A literature review was performed and the following conclusions were drawn:

- Of the various disciplines that have produced research literature on cooperation, the majority of the theoretical advances have been made by biologists, mainly behavioural ecologists/ethologists and molecular biologists/geneticists, their work sometimes drawing on game theory, most notably for the concepts of an evolutionary stable strategy and reciprocal altruism.

- As expected, in computer science, almost all research into the use of cooperation in computing systems has come from the artificial intelligence literature, mainly from the evolutionary computing, distributed artificial intelligence and machine learning sub-disciplines.

- Within the artificial intelligence field the literature concerning cooperative research can be placed into one of two categories, depending upon whether or not it is based on evolutionary computation techniques. The literature review revealed that there is little cross-fertilisation of research between these two categories.

As an immediate consequence of the literature review it was possible to make some recommendations concerning the design of cooperative systems based on standard evolutionary computing techniques such as genetic algorithms and genetic programs, as discussed in Section 3.1.5.

After the literature review had been completed a narrower research focus was needed. In an attempt to identify a more specific focus area it was decided that a number of questions that had arisen during the literature review ought to be considered and experiments designed to help in answering them. It was hoped that this work would provide an insight into which of the questions were likely to lead to productive research. The questions identified were:

- Does the underlying topology of an environment affect the emergence of cooperation in any way?
- What are the likely effects on cooperation of the basic behavioural characteristics of computer systems based on simple evolutionary techniques such as genetic algorithms?
- If reciprocal altruism is to be used as a mechanism for cooperation, how do altruist strategies change as they evolve over time?
- If kin selection is to be used as a mechanism for cooperation, how close does the genetic link between kin recognition and altruism have to be?
- Are there any other mechanisms by which cooperation might be encouraged within a computing environment?

This approach resulted in a number of experimental designs and programs, including:

- A testbed for an investigation into environment topology along with some associated analysis.

- A series of experiments concerning the behaviour of simple genetic algorithm systems (mainly concerning convergence and deceptive functions).

- An experimental design for an investigation into the way that the learning of strategies by artificial neural networks for the repeated Prisoner's Dilemma changes those strategies over time.

- Experiments into the way in which kin selection is affected by inaccurate kin recognition.

- A series of experiments concerning the evolution of a system of economics to encourage cooperation within a population of computational agents.

Out of all of these research threads the one that seemed most promising was the investigation into kin selection, primarily because it highlighted a new consideration for cooperative systems based on kin selection: the problem of kin selective instability.

As mentioned in Section 2.2, it was also decided at this time, after careful consideration, that the evolutionary computing research field would be the area most likely to provide significant progress in cooperative computing, since it is already known that cooperative systems have evolved in Nature (and thus, that it is possible to evolve cooperative systems in this way), and since every symbolic cooperative framework stems from an analysis of the problem domain it would seem preferable to allow a generic system to evolve the required framework, rather than having to analyse every application area in detail before one could be designed. Consequently, the emphasis for research was placed on evolutionary systems.

With the research emphasis on evolutionary systems and as a result of the experiments concerning kin selection, which had highlighted the instability of kin selection, the next stage of research was directed towards answers for the following questions:

1. Since the instability in kin selection seems to be caused by genetic operators such as crossover and mutation, under what conditions will these effects be most pronounced?

2. Are there any mechanisms that can be used to produce stable kin selection?

3. Are there any other mechanisms for cooperation that can be used to encourage the evolution of cooperation in computational environments?

The first question has produced both theoretical and experimental evidence for the link between an increase in the rate of change of an environment, and the decrease in the amount of cooperation within that environment. This work has also identified a new type of selective pressure in fast changing environments and has produced a proposal for the use of a 'sum XOR' or Hamming fitness term as a replacement for mutation-based population diversity (see Chapter 4).

The second and third questions have been studied, and of several promising mechanisms for stabilising kin selection – multiple loci altruism, inter-demic selective systems and group fitness – it has been the last that has produced the most promising results (see Chapter 5).

It appeared that stable, kin selection based cooperative systems could be

developed, and it was proposed that the research topic should be defined as a search for the answer to the following question:

> 'If kin selection is inherently unstable, where is the stabilising factor; and what alternative mechanisms can be used to encourage cooperation to evolve in computational environments?'

It was proposed that a successful answer to this question was attainable, and also that it would constitute an original contribution to the understanding of cooperation within computing systems.

Chapter 2

Overview of Previous Work

The overwhelming majority of research into cooperation has come from biologists. Before any computer scientist attempts to tackle the problems associated with cooperative computer environments it is necessary for him (or her) to understand the relevant biological literature; and in order that the literature may be understood, a grounding in cell biology, genetics, behavioural ecology and other related disciplines is often required.

The following overview is divided into two sections: the biological literature (including the necessary background reading for computer scientists without the relevant biological knowledge), and computing and other literature (including work from areas such as game theory, economics and political science).

2.1 Biological Literature

Although many of the more popular biological source documents are accessibly written and often include useful glossaries, the majority of the theoretical research papers require the reader to be familiar with many biological terms

and concepts. Some good sources for background reading include Alberts, Bray, Lewis, Raff, Roberts & Watson (1989) for information about the molecular biology of the cell, including a wealth of information on genetics; for a more biochemical approach, both Rose (1991) and Holzmüller (1981) are good sources. For an interesting historical perspective Schrödinger (1944) is a physicist's approach to molecular biology before the rôle of DNA was understood. It contains a number of original insights into the foundations of living systems. An associated book containing contributions from many of the current researchers in the field is Murphy & O'Neill (1995). For an introduction to behavioural ecology Krebs & Davies (1991) is the standard authority.

Of the biologists whose research relates directly to cooperation, perhaps the best known is Richard Dawkins, whose work builds on the great theoretical biologists of the past, including Darwin (1859), Spencer (1864) (who coined the phrase 'survival of the fittest') and Fisher (1930), the statistician whose mathematical underpinning of genetics is invaluable. Dawkins' most famous work concerns his 'selfish gene' approach to Darwinian evolution, which considers the gene as the fundamental unit of selection (Dawkins 1976). Since the most successful genes will be those that reproduce most effectively it would at first appear problematic that the natural world contains many examples of organisms cooperating. Although one possible explanation is the 'good of the species' theory of Wynne-Edwards (1962) (also referred to as group selectionism), this would tend to contradict the assertion that natural selection acts on genes, not individual organisms (and definitely not reproductive groups of individuals). Fortunately, there is an alternative theory that explains cooperation, which is based on the work of Fisher and Haldane (Fisher 1930, Haldane 1932, Haldane 1955), and developed by Hamilton

(1964). It is called kin selection and is described in detail in Section 3.1.

The other major contributor to the biological understanding of evolution and cooperation is John Maynard Smith. Of most relevance is his paper on the evolution of social behaviour (Maynard Smith 1982*b*), in which he outlines a possible classification of behavioural models. Maynard Smith argues that any model will consist of one or more of the following mechanisms for social behaviour (taken from Maynard Smith (1982*b*)):

1. *Individual selection.* Not involving selection for altruistic traits[1].

2. *Interdemic selection.* Reproductively isolated groups exist:

   (a) depending on differential production of migrants (Wright 1945),

   (b) depending on group extinction (Wynne-Edwards 1962).

3. *Kin selection.* Reproductively isolated groups need not exist, but interactions occur between relatives (Hamilton 1964).

4. *Synergistic selection.* Interactions occur between non-relatives (Cohen & Eshel 1976, Wilson 1975, Matessi & Jayakar 1976).

5. *Reciprocal altruism* (Trivers 1971).

Interdemic selection can be understood as a mechanism by which 'islands' of individuals prosper if they cooperate; selfish islands of individuals die out more readily and are replaced in the environment by more cooperative

[1]Maynard Smith states: 'By an "altruistic" state is meant a trait which, in some sense, lowers the fitness of the individual displaying it, but increases the fitness of some other members of the same species.'

colonies (as they are more likely to survive). Synergistic selection is a more general form of mutualism where the interacting individuals are not restricted to belonging to different species. For more information on evolution by association in general, see Sapp (1994). Reciprocal altruism includes repeated episodes of interaction characterised by theoretical games such as the 'Repeated Prisoner's Dilemma', invented in about 1950 by Merrill Flood and Melvin Dresher (Axelrod 1984, p. 216), and formalised by Al Tucker shortly thereafter (Nasar 1998, p. 118). The Prisoner's Dilemma is discussed in more detail in Section 2.2. Maynard Smith's paper is important because it lays out, from a biological perspective, the theoretical foundations for the evolution of cooperation.

The other main contribution, by the same author, of relevance to cooperative systems is Maynard Smith (1982*a*), in which the concept of an 'evolutionary stable strategy' (ESS) is explained. According to Maynard Smith, an ESS is a strategy such that, 'if all the members of a population adopt it, no mutant strategy can invade.' Other works of interest include Maynard Smith (1958), Maynard Smith (1989) and Maynard Smith & Szathmáry (1995).

There are many other biologists who have contributed to our current understanding of cooperation but they have not significantly added to the work mentioned above. However, one final mention ought to be made of the contribution of Sewell Wright, who originally defined the 'fitness landscape', a concept that is as ubiquitous as it is useful (Wright 1931).

Although not strictly relevant, it is worth noting that evolutionary biologists fall into one of two camps: the neo-Darwinists (also referred to as ultra-Darwinists), represented by Dawkins, Maynard Smith and others, and

the punctuationists, who support the view that evolution is not gradual but rather proceeds in 'spurts' that punctuate long periods of evolutionary stability. The theory of punctuated equilibria was first proposed by Eldredge & Gould (1972). The punctuationists point to abrupt changes in the fossil record over geological time, which the neo-Darwinists suggest are simply the result of a variation in the rate of evolutionary change. The debate continues.

2.2 Computing and Other Literature

One area of research that has had a major contribution to the study of co-operation is game theory. Although there were previous contributors, for example Cournot (1838), who used a restricted form of a Nash equilibrium (see below), and Zermelo (1913), who proved that chess has only one individually rational payoff profile in pure strategies, the standard work that forms the foundation of modern game theory is von Neumann & Morgenstern (1944), in which two-person, zero sum theory and the notion of a cooperative game with transferable utility are explained.

In the early fifties, John Nash provided important theoretical advances in game theory, notably his proof of the existence of a strategic equilibrium for non-cooperative games – the Nash equilibrium (Nash 1950*b*, Nash 1951), and his foundation for axiomatic bargaining theory (Nash 1950*a*, Nash 1953). More detail on Nash's contribution to game theory can be found in Nasar (1998).

Many developments of game theory have since been produced, including a fascinating theory of convention design for use in automated negotiation

among computers (Rosenschein & Zlotkin 1994). The approach taken uses the analytical techniques of game theory and decision analysis and applies them to the dynamic organisation of autonomous intelligent agents.

For a good, general introduction to classical game theory that covers both static and dynamic games, with or without complete information, Gibbons (1992) is highly recommended.

One of the more recent developments in game theory is a controversial extension called drama theory (Howard, Bennett, Bryant & Bradley 1993, Howard 1994, Bryant 1997), which attempts to tackle the situation where players are not necessarily rational. For example, consider one of the standard game theoretic problems: the Prisoner's Dilemma, which can be described as follows:

> Two people are arrested and separately questioned about a crime that they committed. They can cooperate with each other and say nothing, or defect and tell all.

The consequences can be summarised as in Table 2.1:

Table 2.1: A typical payoff matrix for the Prisoner's Dilemma in which both $S < P < R < T$ and $2R > T + S$ hold

	He cooperates	He defects
I cooperate	3	0
I defect	5	1

Note that for the Prisoner's Dilemma both $S < P < R < T$ and $2R > T + S$ must hold, where S = Sucker's payoff, P = Punishment for mutual defection, R = Reward for mutual cooperation, and T = Temptation to defect.

It is well known that the best strategy is to defect (since the payoff is greater than for cooperation, whatever your opponent does). However, in real life, if best friends were to play they may well decide to mutually cooperate. Their attitudes to each other affect the game. This is less like a pure game and more like a drama.

Another example (taken from Evans (1998)) is the problem of when to get angry. Consider two possible strategies: 'Old Faithful' and 'Mad Dog'. If you have a fixed anger threshold and only get angry if an insult exceeds some predetermined level of annoyance then others can quickly learn what they can get away with, without the geyser exploding. You are constantly pushed to the limit. Alternatively, if you adopt the 'Mad Dog' strategy you choose your anger threshold at any moment completely at random – sometimes a big insult doesn't generate an angry response, at other times a small insult receives the full force of your anger. 'Mad Dog' works better than 'Old Faithful' because others know that even the slightest insult can enrage you, yet you do not have to waste time and effort punishing every small insult. Drama theory is an attempt to extend game theory to take into account these various emotional strategies.

The literature associated with agent systems can in general be characterised as either symbolic, or as evolutionary (although some of the work with artificial neural networks might not fit easily into either category), with little cross-fertilisation other than the adoption of some common problem

domains and assumptions. As has been mentioned in Section 1.1, it was decided that the evolutionary field would be the area most likely to provide significant progress in cooperative computing, since it is already known that cooperative systems have evolved in Nature, and, since every symbolic cooperative framework stems from an analysis of the problem domain, it would be preferable to allow a generic system to evolve the required framework. Consequently, the emphasis in this research is on evolutionary systems. For more information on symbolic agent systems the reader is referred to Haddadi (1996) and Wooldridge, Müller & Tambe (1996).

Perhaps the best known approach to evolutionary computation is the genetic algorithm (GA), first developed by Holland (1975). There are other varieties of evolutionary computation, for example the evolution strategies, which are covered in Schwefel (1995), but the overwhelming majority of the literature concerns GAs.

A standard genetic algorithm typically consists of a population of fixed-length, binary strings that represent candidate solutions to the problem under investigation, together with a suitable fitness function. The pseudocode for a typical genetic algorithm is shown in Figure 2.1.

The termination criteria can include a particular generation number, the existence of a solution with acceptable properties, a measure of the state of the current population (normally a convergence measure), or any combination of the above. As well as Holland (1975), other good, introductory texts on genetic algorithms include Mitchell (1996), Vose (1999), Goldberg (1989) and Michalewicz (1994). Reeves (1995) includes a good chapter on GAs, which covers the theory, various extensions and modifications, and

Figure 2.1: Pseudocode for a typical genetic algorithm

1. Create random initial population.
2. Evaluate fitness of each member.
3. If finished STOP.
4. Create next generation using fitness proportionate reproduction.
5. Modify next generation using crossover and mutation.
6. Replace current generation with next generation and goto step 2.

some interesting applications. Davis (1991) contains useful information on the application of GAs to a variety of problem domains.

In order that the techniques used in genetic algorithm research do not remain restricted to problems whose solutions can be represented by fixed-length strings, several developments have emerged, including the use of hierarchical candidate solutions, most famously as LISP programs in the work of Koza (1992) and subsequently in Koza (1994), and as combinatorial hierarchies in Watson (1994). The use of variable-length GAs occurs in the concept of a 'messy' GA, as developed by Deb & Goldberg (1991).

One of the classic experiments involving cooperation and evolutionary computing was carried out, not by a computer scientist, but by political scientist: Robert Axelrod (1984). The experiment[2] involved the submission of algorithms that implement a strategy for playing the repeated Prisoner's Dilemma (where the number of rounds is uncertain). A computer tournament

[2]Specifically, it was Axelrod's second experiment in which he considered the evolution of strategies for playing the repeated Prisoner's Dilemma.

was then held to determine the best strategy. Although the strategy 'TIT FOR TAT' did well, one of Axelrod's conclusions was that the best strategy depends upon the other strategies within the population.

An extension of Axelrod's work has been produced by Lomborg (1996) in which mutation, implemented as a faulty copying of the strategy from parent to offspring, introduces noise into the environment. Lomborg maintains that the system evolves unexpected stability as a result of the noise, ensuring that the chances are low that the introduction of a new strategy will result in it dominating the population.

There are many more papers to do with the Prisoner's Dilemma, for example Oliphant (1994) and Mor, Goldman & Rosenschein (1996), but most add little more than has already been discussed.

One other major source of relevant research is the artificial life literature (Langton 1989, Langton, Taylor, Farmer & Rasmussen 1992, Langton 1994, Brooks & Maes 1994, Varela & Bourgine 1992). In particular, the work by Hillis (1992) into co-evolution is highly relevant to all symbiotic environments. Hillis shows that by co-evolving the test cases in parallel with the sorting functions in which he is primarily interested, if the fitness of a test case is based upon its ability to cause problems for a sorting function, then the resulting population of sorting functions will be more robust than if the test cases were not co-evolved. Hillis' work has spawned a mini-industry in problem solving through the use of GAs and co-evolution. Examples of the application of co-evolution can be found in Potter & De Jong (1994) and Reynolds (1994). A good, general introduction to artificial life can be found in Sigmund (1993).

There are many other papers, journals and sources that contain material relevant to the study of cooperation but the overview presented here should give the reader a feeling for the current state of related research. Further references to literature specific to particular topics are provided in the thesis in the relevant sections.

In the next few sections some aspects of cooperative environments will be considered and some new insights will be explained – starting with the consequences, for cooperation, of kin selection.

Chapter 3

The Instability of Kin Selection

Modern explanations of cooperation are based on two main theoretical models: kin selection and reciprocal altruism. If individuals learn to cooperate within a particular generation (for instance, when playing a version of the repeated Prisoner's Dilemma) it is reciprocal altruism that is used to explain their behaviour. Alternatively, when there is little or no learning within a single generation and when cooperation persists from one generation to the next, the standard explanation relies on kin selection. For cooperative, evolutionary computing systems the theory of kin selection is extremely important. In this chapter, kin selection is considered in detail and an experiment is described which shows that kin selection is not a sufficient explanation for the persistence of cooperation.

3.1 Kin Selection Considered

Kin selection can be summarised with the statement: 'By helping a relative, an individual is propagating its own genes.' In his most famous paper, Hamilton (1964) formalised the notion of kin selection and introduced the

concept of a criterion for kin selective dominance, known as Hamilton's inequality. Later work by Hamilton showed that kin selection should still work for genes irrespective of their rarity. Hamilton's inequality can be formulated as follows (adapted from Maynard Smith & Szathmáry (1995, p. 259)):

Imagine a rare gene A, which causes an individual D (donor) to perform an act X. The effects of act X are:

- To reduce the expected number of gametes that D passes on to the next generation by c (cost).
- To increase the expected number of gametes that a 'recipient' R passes to the next generation by b (benefit).

Consequently, Hamilton's inequality states that act X increases the number of A genes in the next generation if:

$$b/c > 1/r$$

where r is the coefficient of relatedness between D and R.

As was stated above, kin selection is now considered to be the major inter-generational mechanism for the evolution of cooperation (the major intra-generational mechanism being reciprocal altruism). Before considering kin selection in any more detail, it is necessary to explain it carefully, and to point out the main sources of confusion. The following explanation is adapted from Watson (1996).

3.1.1 The Need for Kin Selection

Evolution is seen, by neo-Darwinists, as a process controlled by three factors: mutation, natural selection and migration (Maynard Smith 1989, p. 180). Within a given environment a population of replicators generates novelty through mutation, develops adaptation with natural selection, and produces new, distinct populations through migration (called allopatric speciation). The main difference between Darwin and the neo-Darwinists is that Darwin thought of reproduction as a blending of parental information, as opposed to the modern, particulate theory of genetic recombination (Darwin 1859, p. 47). But the real difference is that Darwin thought that natural selection selected the fittest *individuals*, and the neo-Darwinists believe that it is the gene that is the true unit of selection. The individual is seen by them as merely a vehicle for the replicating genes (Dawkins 1976, p. 254). For a good explanation of the molecular biology involved, see Alberts et al. (1989) and Rose (1991); for an investigation into some of the consequences of neo-Darwinism, see Jones (1993) and Gould (1980).

The modern theory of evolution is very good at explaining the complex adaptations observed in living systems, for example: sophisticated organs such as the eye, various forms of animal mimicry, and parallel adaptations of predators and prey. But, by concentrating on the gene as the unit of selection, some aspects of life seem difficult to explain. Examples of these include the development of sexual reproduction, the observed rates of mutation within genetic material and the apparently altruistic behaviour of some individuals towards their close relatives (which includes the phenomenon of parental care). All three of these examples have been explained by recourse

to the 'good of the species' (for an intelligent example of this type of explanation, see Wynne-Edwards (1962)), but group selection (as it is known) is itself something that seems opposed to a view of genes as 'selfish' replicators, with no ability to see into the future. Superficially, from the point of view of an individual gene, sexual reproduction is only half as effective as parthenogenesis[1], no mutation is better than some mutation[2] and a selfish vehicle seems to be the best form of transport. Kin selection is a theory, first suggested by W. D. Hamilton (although Maynard Smith (1958, p. 195) claims to have suggested the term 'kin selection'), which attempts to explain the apparently anomalous, altruistic behaviour of certain individuals, without recourse to group selectionism (Hamilton 1964).

3.1.2 Altruism and Kin Selection

In order to explain the principles behind kin selection it is useful to talk of genes as though they consciously choose between a number of alternatives, and to think of them as promoting or inhibiting various phenotypic phenomena. Thus, during this explanation of kin selection, it will be useful to state that a gene chooses to cause an individual to act altruistically. It is worth stating, to allay any possible misunderstandings, that genes do not

[1] Reproduction from a single germ cell (a gamete) without the need for fertilisation – thus passing on all of the mother's genes to the daughter.

[2] Although it may seem that mutation is fixed by the environment, in fact there are populations of bacteria that contain individuals with repair enzymes that are more accurate than the population average – leading to these individuals having a lower mutation rate. However, over successive generations, the population average mutation rate doesn't go down, so it would appear that the mutation rate is chosen by the bacteria (Maynard Smith 1989, p. 184).

have free will or intelligence, and that genes do not wholly determine characteristics such as altruism. In spite of this, it is readily acceptable that, in a given environment, two individuals, differing by one gene, might display slight differences: a slightly longer leg in one, or a slight change in brain development such that, all things being equal, one individual might be slightly more cowardly than the other – or slightly more altruistic. Given a population containing a variety of genes (or, more accurately, gene values) at the same locus on a chromosome[3], and given that they produce functional differences in their corresponding phenotypes, then it is reasonable to assume that some genes will be more likely than others to find themselves in the next generation. In this way, a gene that 'chooses' to promote a particular trait can be said to prosper in relation to other genes, at the same locus, which make inferior choices.

The theory of kin selection can be explained within the context of selfish genes by considering the 'purpose' of a gene – namely, to get the maximum number of copies of itself into the next generation (i.e. each copy of the gene tries to replicate as much as possible). Consider the following: an individual has found a hidden food source providing plenty of food. Should it let others know about it? To answer the question consider what would happen if the others were unrelated. Feeding them would increase their chances of reproduction by increasing their chances of survival. This will increase their probability of mating with a given partner, and thus decrease the discov-

[3]Mutating a gene may alter the phenotype in some way. A subset of all possible mutations of this gene will exist in the population and are known as alleles – alternatives at a given locus. These alleles can be thought of as competing genes, with the alleles that produce the most successful phenotypic variants being considered the winners within the population.

erer's chances of doing the same (all things being equal). So selfishness is the best strategy. In a given population, with these circumstances the selfish individuals will prosper, along with the genes that predispose them towards selfishness.

Now consider the case where all the individuals are closely related. If they are brothers, from the same pair of parents, then they will each contain half of the discoverer's genes (on average). There is plenty of food so feeding them will increase the number of copies of those genes in the next generation. The more closely related they are, the greater the advantage conferred on the discoverer's genes. So it may well pay to be altruistic. This is kin selection.

Before individuals can behave altruistically towards their close relatives, they must first recognise them. In other words, a gene that promotes recognition of kin will allow altruism to develop. Thus, some smells, calls, patterns of movement and markings can be explained in terms of kin selection. But what about a situation in which altruism involves a far greater sacrifice? Should an individual sacrifice itself for one brother, or two, or three, and how about cousins? In order to answer these questions a quantitative measure of altruistic advantage is needed. The conventional measure is called inclusive fitness.

3.1.3 Measuring Altruism

Fitness is both a useful and a confusing concept. In Darwin's time, fitness was considered to be linked to the utility of an individual's organs. An animal with a stronger jaw, or a sharper eye, is more likely to survive and thus to

reproduce. In time, it was seen that the link between fitness and survival was so useful that fitness was redefined to mean, 'the factor which predisposes an individual to be likely to survive'. This definition of fitness rendered Herbert Spencer's phrase, 'survival of the fittest', tautologous (Spencer 1864). The advent of the 'modern synthesis' (as neo-Darwinism is often called) brought a new definition of fitness, emphasising the importance of the gene as the unit of selection. This is what is referred to by the term *inclusive* fitness.

Hamilton (1964) realised that classical fitness is too limited in scope: it is a measure of the reproductive success of a single individual. But an individual is merely a vehicle for the real replicators – the genes. The reproductive success of an individual is irrelevant, since it is ephemeral. What is needed is a measure of the reproductive success of genes. However, this measure has to be associated with individuals for it to be useful in practice, as in nature we perceive individuals, not genes. Consequently, Hamilton defined inclusive fitness to be an individual's own reproductive success plus its *effects* on the reproductive success of its relatives, each one weighted by the appropriate coefficient of relatedness – half for each brother, one-eighth for each cousin, and so on.

3.1.4 Confusions of Inclusive Fitness

An important point is that if a brother emigrates to Australia, then the individual's inclusive fitness isn't increased every time its brother reproduces, as the individual can have no effect on its brother's reproductive success (barring exceptional circumstances). Thus, the fitness of an individual is related to the effectiveness of its genes to get themselves replicated in future gener-

ations, i.e. how effective a gene is at replicating itself, plus what difference it makes to the replication success of the gene at the same locus in all relatives, weighted by the probability that these loci contain copies of itself.

Notice also that this definition of fitness is a relative measure, as opposed to absolute, classical fitness, and that inclusive fitness is a property of a triple consisting of the individual in question, an act or set of acts of interest, and an alternative set of acts for comparison. A common fallacy is that inclusive fitness is the weighted sum of the reproductive successes of the individual and all of its relatives. One of the problems associated with this view is that children can often be counted several times, as though they have many existences. The various, common misunderstandings of kin selection are discussed in Dawkins (1979), and for the interested reader, a good description of the different meanings attributed to fitness is provided by Dawkins (1983, pp. 179–194).

3.1.5 Designing Cooperative Environments

So what are the requirements for the emergence of kin selective effects in a computer environment? First, the individuals (software agents) need to be active – it is not sufficient to have a standard genetic algorithm model of evolution, modified to increase interactions between relatives, and to expect that kin selection will produce cooperation between individuals. There has to be some way for the *behaviour* of an individual to have an effect on the reproductive success of its relatives (i.e. with a standard GA, an individual's fitness must be influenced by the behavioural effects of its relatives). A more suitable model for active individuals would appear to be Koza's genetic

programming (GP) approach (Koza 1992, Koza 1994). However, the rôle of chromosomal loci should be considered, since GP individuals do not generally contain homologous chromosomes.

A locus on a chromosome in many respects defines the function of a gene. When an organism develops and its cells divide and differentiate, different parts of the body contain cells with different gene expression. In other words, the bits of DNA that are unravelled and exposed to the cellular mechanisms are different for the various parts of the body. Thus, the various gene values that affect eye colour will be found in the same loci. But GP uses tree structures that vary widely and which don't have loci. However, if the genetic units within a GP structure continue to perform the same function as they travel through a series of generations, then they would be behaving as though they have loci. Unfortunately, this isn't generally the case. This leads to another consideration, namely the function of a genotypic unit (a gene) in the context of an individual.

When thinking about evolutionary adaptation it is easy to get confused by the difference between the function of the individual and the gene. Evolutionary pressures affect genes, causing the best reproducers to dominate the population. Genes group together and form collective, phenotypic entities – individuals – because this gives them the best reproductive advantage (more is said on this subject in Chapter 5). A particular gene contributes in a specific way to the individual's overall phenotype; perhaps it affects eye colour, or whether it shares its food, or perhaps it does both. A gene which contributes to more than one phenotypic trait is called pleiotropic. For a useful trait to dominate the population (for example, long legs) there must be a particular collection of gene values in the population that affect leg length to

a greater or lesser extent. Then, since longer legs mean that individuals will be better at reproducing, the population will be dominated by long legged individuals, all carrying a 'long leg' gene value. For kin selection to affect the population it is wrong to think that altruistic animals will dominate because they will help each other and thus become more reproductively efficient. This will only happen if there is one gene that simultaneously encourages altruism, and that helps in the recognition of relatives. This is because evolution works on genes, not on individuals. Dawkins gives an example of a gene which causes beards to become green and that also encourages altruism (Dawkins 1983). It may be that two closely linked genes are affected by a weaker kin selection, since they will appear to be one large gene[4], and the larger the gene, the more likely it is to be broken up by genetic crossover, thus the advantage gained through being better at reproducing is reduced by its higher likelihood of being broken up.

Consequently, for kin selection to work, we need to have pleiotropic genes that simultaneously encourage altruism and increase the chances that an individual will be altruistic to close relatives. This can be done by recognising relatives by their traits, or by their physical location in a metric space (when, given suitable reproductive and migratory behaviour, close relatives will tend to be the nearest individuals). Examples of models that use the concept of locality can be found in the artificial life literature (Frean & Abraham 2001, Langton 1989, Langton et al. 1992). Furthermore, these genes need to

[4]Genes are not strictly defined. Theoretical approaches to genetics usually assume genes to be atomic so, as progressively larger collections of codons – or bits – are considered, the more likely it is that such a collection will not survive intact across a number of generations, thus making them appear less like atomic genes. However, it is not unreasonable to talk of two closely linked genes as appearing to be a single, large gene.

perform a similar function as they travel through generations of individuals. So, for kin selection to work, a computer environment will need to contain active individuals with functionally constant, pleiotropic genes, and perhaps would benefit from the inclusion of an environment which promotes neighbourhoods of close relatives.

3.2 An Experiment to Test the 'Green Beard' Effect

Although the theory seems to be convincing, since the effectiveness of any cooperative computing system based on kin selection will rely on some form of kin recognition[5], it seemed sensible to confirm the theoretical results experimentally. Consequently, an experiment was designed to test the 'Green Beard' effect of kin selection, as described by Dawkins (1983, pp. 143–155).

If genes for growing a green beard and for helping green-bearded individuals occur sufficiently close to each other on a chromosome then, as long as

$$\frac{\text{benefit to recipient}}{\text{cost to altruist}} > \frac{1}{P(\text{recipient has altruist gene})}$$

it follows that the altruist gene should be advantageous and should dominate the population.

The experimental model of the chromosome can be thought of as an N bit string containing two, one bit gene loci: a 'green beard' gene locus and

[5]Note that kin recognition is present but implicit in models where relatives tend to be neighbours in a spatially extended environment.

an 'always help a green-bearded individual' gene locus. A one in either locus indicates that the gene is active. The experiment uses one point crossover, which results in the genes being disconnected and reconnected with a new gene if the crossover point falls between the two gene positions. The probability of this happening is

$$\frac{\text{number of crossover positions between loci}}{\text{number of crossover positions in chromosome}} \quad = \quad P_c$$

If the genes were adjacent in an infinitely long chromosome they would be inseparable ($P_c = 0$). If they were at either end, then $P_c = 1$. Halfway between these extremes: $P_c = \frac{1}{2}$. Irrespective of the length of the chromosome, $P_c = 0$ can be considered to represent a single, pleiotropic gene. Thus the population of chromosomes can be modelled by a collection of two bit strings with a single, fixed P_c value.

The experiment should be able to confirm that the altruism gene is advantageous if

$$\frac{b}{c} \quad > \quad \frac{1}{P(\text{recipient} = 01 \text{ or } 11)} \quad = \quad \frac{\text{number of 11s and 10s}}{\text{number of 11s}}$$

where b is the benefit to the recipient, c is the cost to the altruist, and where the first bit in a bit pair represents the green beard gene, with the second bit representing the help green beards gene. The ratio on the far right shows that only 11s will receive any benefit (since 01s don't have a green beard) and that, similarly, 00s and 01s are never the recipient in any altruistic exchange.

Several experimental runs were performed (see Appendix A), each with a variety of initial populations – for example, half 00s and half 11s ($\frac{b}{c} > 1$), or the same number of each of the four possibilities ($\frac{b}{c} > 2$) – and with different values for $\frac{b}{c}$. Over several generations the population was randomly split into

pairs, the first in each pair would then interact altruistically dependent upon its second bit value, then the next generation was formed and, if not turned off, the genetic operators, crossover and mutation, were applied.

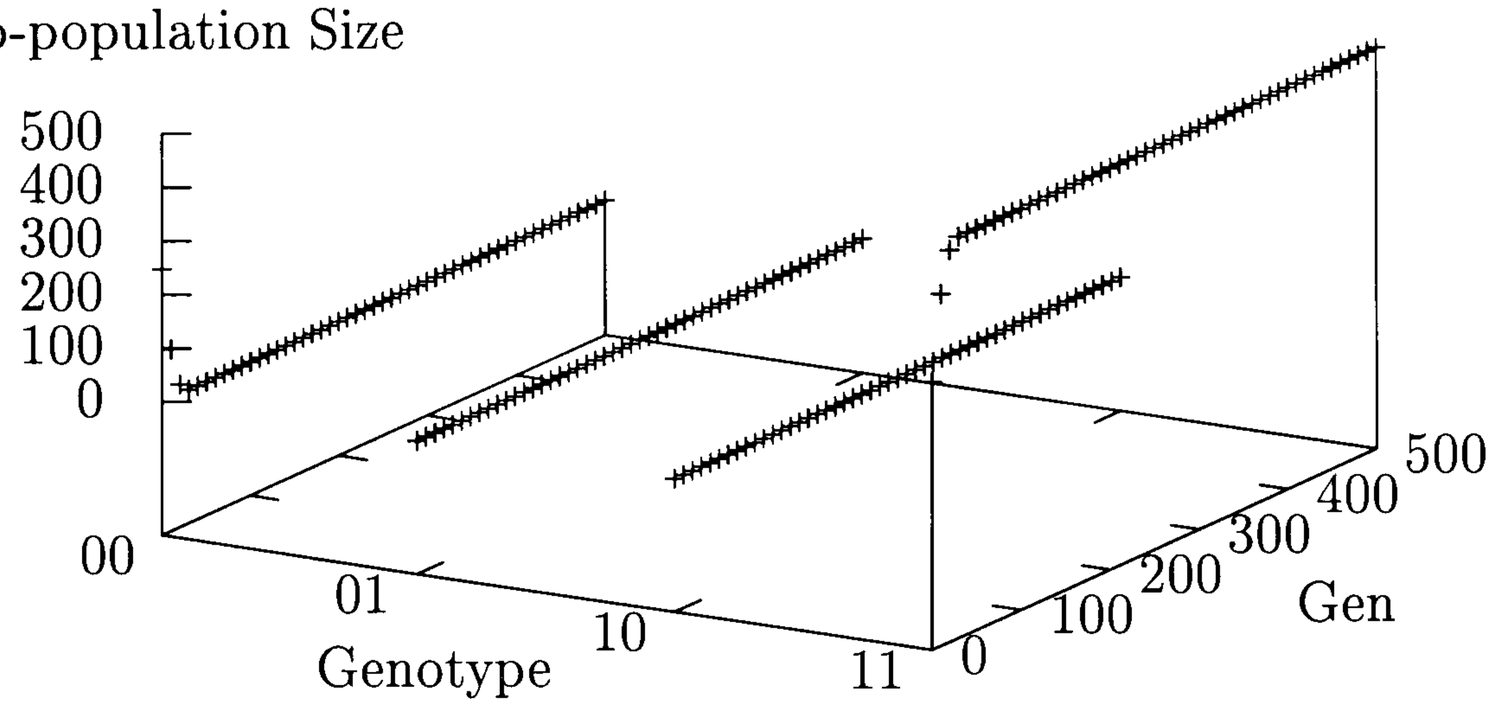
Two points concerning the program are worth highlighting. Firstly, as in all experimental programs that rely on simulating random processes, a good pseudo-random number generator (PRNG) is important. The PRNG used for this research is included in Appendix H. Secondly, this program, and some of the later ones, use an interesting method of simulating a random shuffling of the population when all that is required is to randomly partition a population into pairs. This method is covered in Appendix D.

3.2.1 Results and Conclusions

With the crossover probability P_c equal to 0 (in effect, simulating a single, pleiotropic gene) and with no mutation the results were as expected when the initial population contained no 10s. Figure 3.1 shows that an initial population of roughly half 00s (selfish, no green beard) and half 11s (altruistic, green bearded) is quickly dominated by the altruists when it pays to be altruistic (see also Appendix A.2). When there is no benefit but still a cost associated with altruism, Figure 3.2 and Appendix A.3 show that the selfish individuals dominate (the fluctuations are the result of stochastic effects when the altruists become so rare that they almost never meet each other and consequently almost never behave altruistically). But for even extremely low values of P_c and/or mutation, and for initial populations that contained 10s, the altruism gene would never become dominant. Figure 3.3 shows the results for a typical experiment that represents individuals with

a single, pleiotropic, altruistic green beard gene (i.e. no crossover) and low mutation (see also Appendix A.4). This result should be expected, since kin mimics (10s) will reap the rewards of altruism, but won't suffer any of the associated costs. One point to note is that as the kin mimics increase in frequency their selective advantage decreases since there are fewer altruists about. At a point arbitrarily near to complete population convergence to all kin mimics, the variation in numbers of kin mimics is arbitrarily close to a random walk.

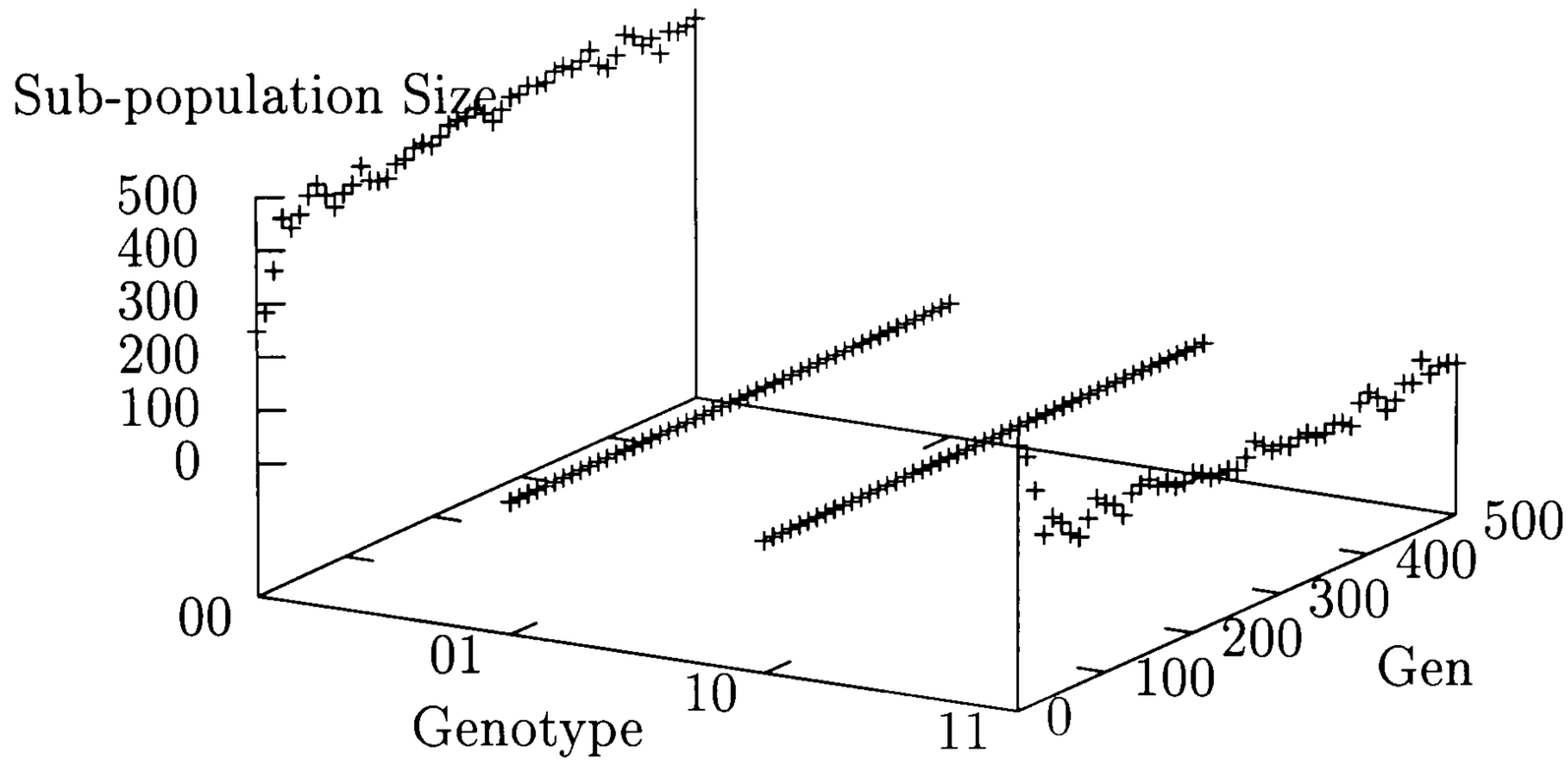
Figure 3.1: Results for 'Green Beard' effect program with no crossover and no mutation



The consequences of this experiment for kin selection are enormous. It shows that kin selection is inherently unstable and, in any population when given enough time, altruistic cooperation will die out. An analysis of kin selective instability for large populations is given in Section 5.1.

A similar result will occur in an evolving population in which genes confer

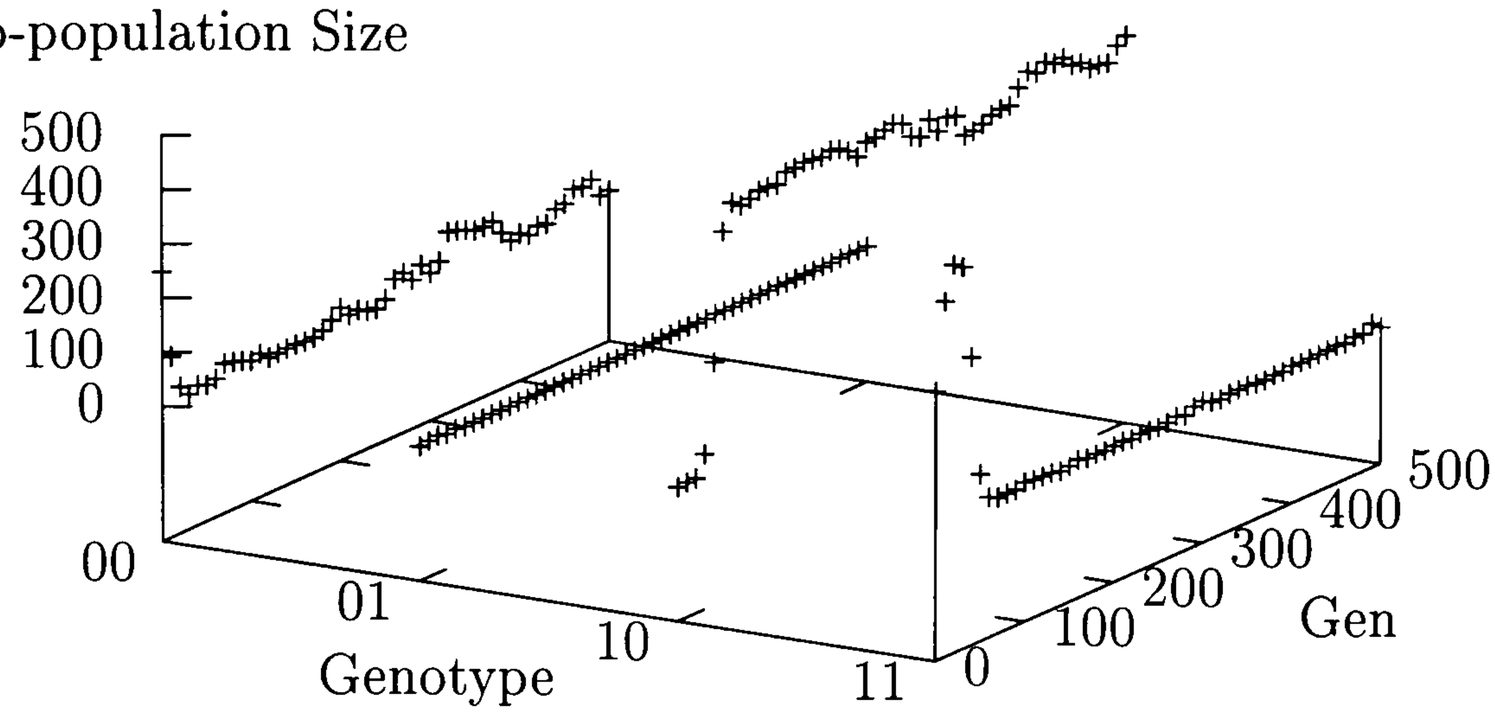
Figure 3.2: Results for 'Green Beard' effect program with no crossover, no mutation and with benefit= 0



differing probabilities of looking like and acting like an altruist. Consider an individual with a particular pair of probabilities (P_l, P_a), where P_l is the probability of looking like an altruist and P_a is the probability of acting like one. If another individual with a different set of alleles has a probability pair (P'_l, P'_a) with $P'_a < P_a$ then it will dominate the first if $P'_l \geq P_l$ or if the extra fitness it gets by being less altruistic more than compensates it for any loss incurred by looking less like an altruist. In this way the population will tend to become dominated by individuals that display the most profitable possible combination of looking and acting like an altruist. While selective pressure favours gene values that promote looking like an altruist evolution certainly doesn't encourage being one.

On rereading the kin selection literature it almost seems as though the biologists understand the problem. In his paper on kin selection Grafen (1984, p. 79) states at one point, when considering preferential assortment

Figure 3.3: Results for 'Green Beard' effect program with no crossover



(relatives grouped together), that,

> 'there would be selection for a "free-rider" allele (if one arose) at a locus unlinked to the altruism locus. It would have the effect of creating the same habitat preference as that of the altruists, whether or not its bearer was an altruist.'

However, Grafen doesn't appear to conclude that this is a potentially fatal flaw for kin selective altruism.

Maynard Smith (1958, p. 199) also seems to have considered the problem. In a section that discusses group selection and altruistic individuals that voluntarily refrain from reproducing when the population density becomes too high he points out that, in a mixed population of altruists and non-altruists, the altruistic individuals will be eliminated by selection. Again, this is not seen as a serious problem for kin selection. In fact, Maynard

Smith suggests that, rather than group selection, it is kin selection that can be employed to explain the persistence of this kind of behaviour in animals such as mice and lemmings.

More recently, in March 2003, in an 'Inside Science' report on cooperation in the 'New Scientist', Randerson (2003) highlights the problems of cheating in cooperative populations. But, rather than cheating being a problem for kin selection, it is the cheaters who have the problem. The end of the report is as follows:

> 'But even in slime mould society there are cheats, because although cells that become spores get to spread their genes far and wide, stalk cells are doomed. Kin selection helps because the spores and stalk cells are related, so the sacrifice by the stalk cells benefits genes they have in common with the spores. However, fruiting bodies often contain cells from two different, less closely related groups, and sometimes one of the groups cheats by contributing more than its fair share of spores, leaving the other group to build most of the stalk.
>
> There can be little doubt: selfish genes are behind both cheating and cooperation. Despite the existence of so much teamwork in the natural world, Darwin's vision of nature red in tooth and claw still holds true, because if you dig deep enough, cooperation is always underpinned by genes looking after number one. But watch out: in nearly every successful team there are freeloaders who are happy to sit back and reap the rewards of everyone else's hard work.'

This seems to be a description of a stable, cooperative population in which cheating is controlled by kin selection.

The instability of kin selection could also explain a curious comment in Ackley & Littman's paper on altruism in the evolution of communication (Ackley & Littman 1994, p. 47). After explaining the emergence of altruistic communication by using kin selective arguments, the authors write:

> 'From an evolutionary computation point of view, perhaps the most striking aspect of these studies is that they make no use of mutation. As mentioned earlier, we were pushed to this decision, against our own preconceptions, simply because it worked better.'

Perhaps the reason is that even a small amount of mutation will give rise to free-riding kin mimics?

Frean & Abraham (2001) are also affected by the instability of kin selective cooperation. They investigate a spatial version of the Prisoner's Dilemma in which strategies for playing the game are placed on a two-dimensional grid. At each time step, a strategy at a particular site is chosen to play with one of its neighbours. The strategy invades its neighbour with a probability that is proportional to the payoff it receives from the game. It might be expected that, since relatives tend to be close to each other, kin selection would encourage cooperation. This appears to happen. However they state that,

> 'the average level of cooperation decreases with time if mutation of the strategies is included. Spatial effects are not in themselves sufficient to lead to the maintenance of cooperation.'

Frean & Abraham offer no explanation of this result but it would appear to be another example of the destruction of cooperation by kin mimics.

Even the most respected of researchers into cooperation would seem to have missed the point. Riolo, Cohen & Axelrod (2001) attempt to show that cooperation can evolve in systems that don't require reciprocal altruism. This is an implicit acknowledgement that previous attempts to do so have not been successful. They evolve a population of individuals, each with two real-valued genes: one gene that represents a 'tag' or identifiable phenotypic trait, the other gene representing a tolerance for cooperation. Each gene value is between zero and one, and if an individual interacts with another that has a tag value tolerably close to its own then it will cooperatively donate some fitness to it. The authors claim that cooperation emerges and is maintained within their model.

In a response to Riolo et al. (2001), Roberts & Sherratt (2002) quite correctly point out that cooperation cannot fail to emerge as the least cooperative individual will have a zero tolerance and will still have to cooperate with clones of itself. In fact the model shows that as the population evolves the tolerance levels drop – indicating that the individuals are doing their best to lower their rate of cooperation. In a modified model, Roberts & Sherratt show that cooperation will die out if allowed to. However they don't conclude that this is because of kin instability. They state that,

'Cooperation under the original conditions of Riolo et al. operates through a process of "like helping like" – agents sharing any particular tag also share the rule of donating to each other, so a form of kin selection can support cooperation. However, agents

can have identical tags without having a recent common ancestor, so in our modified system they can share tags without sharing the rule for cooperating. Because tag similarity is no longer a reliable guide to behaviour, the system of "like helping like" breaks down.'

Proponents of kin selection would agree that if the genes for altruism are not also the genes for the recognised trait then cheating might destroy cooperation. The response from Riolo, Cohen & Axelrod (2002) is not convincing. And even if altruism and recognition were to be controlled by the same genes, it has been shown that kin selection is still unstable.

Since cooperative populations exist in Nature, either kin selection is stable because of some other factor, or the perceived stability of cooperative populations is due to a different mechanism. The question is: where is either the missing factor, or the alternative mechanism? Various candidate solutions (including multiple loci altruism, and group selective approaches to overcoming the 'tragedy of the commons' (Hardin 1968)) have been investigated and Chapter 5 describes how cooperative populations can be stabilised.

But is the instability of kin selection the only reason why cooperation might not emerge in computer-based evolutionary systems? In Chapter 4 another equally problematic phenomenon is considered – the effect caused by an environment that changes from generation to generation – and a mechanism for overcoming it is revealed.

Chapter 4

Coping with Fast-Paced Environments

Cooperation takes time to develop. From the cooperative actions of opposing, first World war soldiers to the formation of business cartels, if the environment does not stay constant for a sufficiently long period then cooperation will fail to emerge. And it is not just reciprocal altruism that suffers from this problem. For kin selection to produce cooperation (ignoring, for the moment, that it is unstable), there needs to be enough time for mutation and crossover to produce sufficient numbers of altruists so that interactions occur between them, and enough time for them to become sufficiently fit, and sufficiently numerous, to dominate the population.

The problem, stated simply, is to find the optimum (or optima) before the optimum changes. In an evolving system in which an individual's fitness is relative to the other members of its generation, the optimum is changing with every generation. Consequently, a fast-paced environment is often a barrier to evolving, cooperative systems.

This chapter considers the difficulties of finding and tracking optima in

fast-paced environments. It then identifies and tests some promising methods for overcoming these problems.

The first topic under consideration is mutation. Since mutation rates seem to be intimately associated with the rate of change of a population, the next section considers the possible advantages to be gained from varying the mutation rate.

4.1 Changing the Mutation Rate

Mutation within evolving systems is often viewed with ambivalence. While it is undoubtedly a good source of novelty, and in spite of the benefits it confers by contributing to population diversity, it nevertheless seems like a necessary evil; and it has been argued that populations will evolve mutation rates that are as low as possible (Drake 1991). But what is mutation?

4.1.1 Mutation Considered

Imagine a simple, evolutionary system with a fixed-size population and chromosome length. Consider a single locus (one bit position) on the chromosome: at any point in time the state of the population for that locus can be represented by the number of ones in that position within the population. Mutation in such a system can be thought of as a spring-like force, dragging the population back to a state of equal numbers of ones and zeros at all loci, with a force proportional to the number of ones at a given time, at each locus.

The standard definition of mutation is that it is a random change within a replicator. Although intra-generational genetic changes can occur, here the concentration is on mutation as a result of transcription errors. One of two consequences is possible from this definition of mutation:

1. If the offspring are not replicators then mutation reduces to random search.

2. With viable offspring and with some form of selective pressure, mutation provides the basis for evolutionary adaptation.

So should mutation always be seen as an unwanted effect? It would seem that if the goal of a successful replicator is to maximise the number of copies of itself in the next generation then it should try to minimise its mutation rate.

In a stable environment, it is true that the most successful replicators are those that produce the greatest numbers of copies of themselves, with the greatest copying fidelity. This is because the offspring will perform similarly to their parents, as the environment does not change significantly from generation to generation. Yet it is known that some biological systems associated with dynamic environments maintain above-minimal mutation rates (Maynard Smith 1989, p. 184). An explanation for this is that, in these systems, increased mutation rates could be used by an organism to react to environmental change either within a generation or between generations. Care must be taken, when considering this explanation, as it is the gene and not the individual that is the fundamental unit of selection.

It has also been shown that within an organism the mutation frequency

can vary by more than a thousand times from gene to gene. Why each gene has its own mutation rate is not known (Jones 1993, p. 87). Richard Moxon, a biologist from the John Radcliffe Hospital at Oxford University, believes that the 'hypermutable' genes generate useful biological noise at the cell surface (Brookes 1998, p. 39). He suggests that most genes perform basic 'housekeeping' functions and have low mutation rates, whereas a limited set of hypermutable 'contingency' genes within the organism are selected to have high rates. The reasoning is that a mistake in a vital housekeeping gene may well be fatal, but mutations in contingency genes might add some form of genetic flexibility. However, this argument is based upon the advantage for an individual, not for a gene. Nevertheless, this raises an interesting related question: could a gene ever evolve that increases the mutation rate in another gene (or several other genes) to maximise the average fitness of its host's offspring in the next generation?

So, could such a gene ever evolve to increase the mutation rate in other genes, and if so, under what conditions? Some work has been done on this question. Taddei, Radman, Maynard-Smith, Toupance, Gouyon & Godelle (1997) considered the role of mutator alleles – gene values that promote high mutation rates – in adaptive evolution and concluded that, within ever-changing environments, mutators can be maintained alongside antimutators (low mutation rate alleles) within the population by a process of 'hitch-hiking' (i.e. benefiting from the selective advantage of a neighbouring gene value). However, their model was based on asexual reproduction and did not support crossover.

The work of Stephens, Garcia Olmedo, Mora Vargas & Waelbroeck (1998) is also relevant as they studied models which showed a rise in mutation

associated with a change in the environment. However, their model was primarily concerned with symmetry breaking and did not consider repeated environmental change.

Other research related to dynamic mutation rates includes Bäck (1992), Nijssen & Bäck (2003) and Grefenstette (1999). While Bäck's work differs from other attempts in its use of bits within the individual's chromosome to encode the mutation rate, thus allowing it to adapt by selective advantage and consequently removing the need for the rate of mutation to be controlled externally, the work only considered static fitness landscapes and was designed to help with the application of GAs by removing the complication of determining the best mutation rate for a given problem. Grefenstette's work is the most similar to the research outlined in this thesis but his focus is on 'hypermutation', i.e. randomly resetting an individual's genotype. However, like Bäck, he also prefers the self-adaptation of mutation rates to externally imposed heuristics. While Grefenstette correctly points out that GAs are expected to be well-suited to dynamic fitness landscapes, his fitness function is far more complicated than the one used in this thesis, consisting of hundreds of sub-optimal fitness peaks and one globally optimal peak all moving randomly over time. Grefenstette also highlights two different interpretations of mutation: that a mutated bit is either flipped – the most common interpretation within the GA literature; or that a mutated bit is reset at random – Holland's original interpretation (Holland 1975). In his work, Grefenstette uses the latter form of mutation. However, since a probability of one that a bit is reset has the same effect as a probability of a half that it is flipped, the distinction, while important, is only one of magnitude. In this thesis, a mutated bit is flipped rather than reset.

The next few sections attempt to show, with the aid of some simple computer models, how increased mutation rates controlled by mutation genes can be advantageous within evolutionary systems with fast-paced environments and how this advantage can be harnessed within computer systems based on genetic algorithms.

4.1.2 A Simple Model of Evolving Mutation Rates

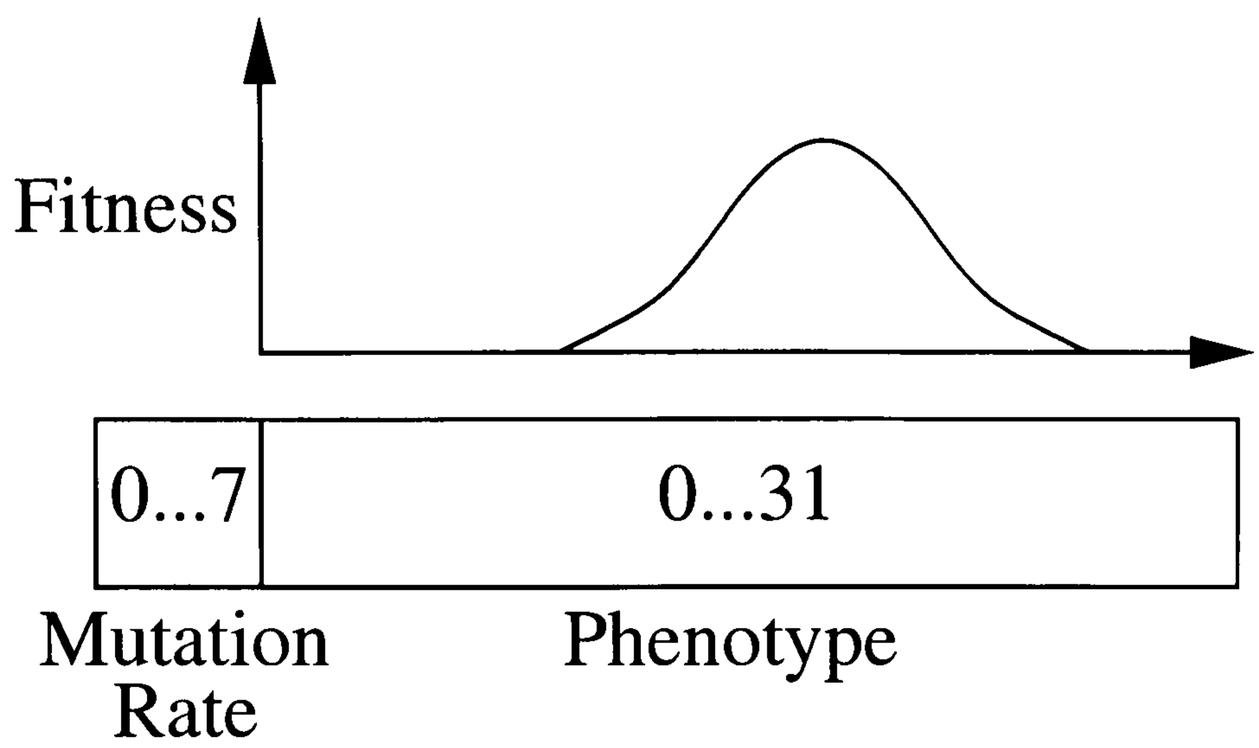
To test whether the increase in some observed populations' mutation rates is adaptive and associated with the speed of environmental change, a simple model was created. Each individual within the modelled population was represented by two integers: a phenotype value $p \in [0 \ldots 31]$, and a mutation rate $m \in [0 \ldots 7]$ (see Figure 4.1). The environment consisted of a normally distributed, dynamic fitness function f_g, defined over all generations $g \in [0 \ldots)$, with a single optimum at f_g^*, where

$$\sum_{p=0}^{31} f_g(p) = 32 \,, \quad \text{and} \quad f_g(p) = f_0(\lfloor (p + kg) \bmod 32 \rfloor) \quad \forall\, p \,,\; g \,.$$

A series of experiments was carried out using a variety of environmental speeds $k \in [1/4 \,, 1/2 \,,\; 0 \,,\; 1 \,,\; 2 \,,\; 4]$, in which a fixed-size population of several thousand individuals was monitored over many generations.

The experiments all started with a uniformly random population. For every generation, the fitness of each individual, determined by its phenotype value, was used to calculate the number of offspring it produced, asexually, in the next generation (i.e. fitness proportionate reproduction with no crossover). The phenotype values of the offspring were determined by their

Figure 4.1: Chromosome format and fitness function in a simple model of evolving mutation rates

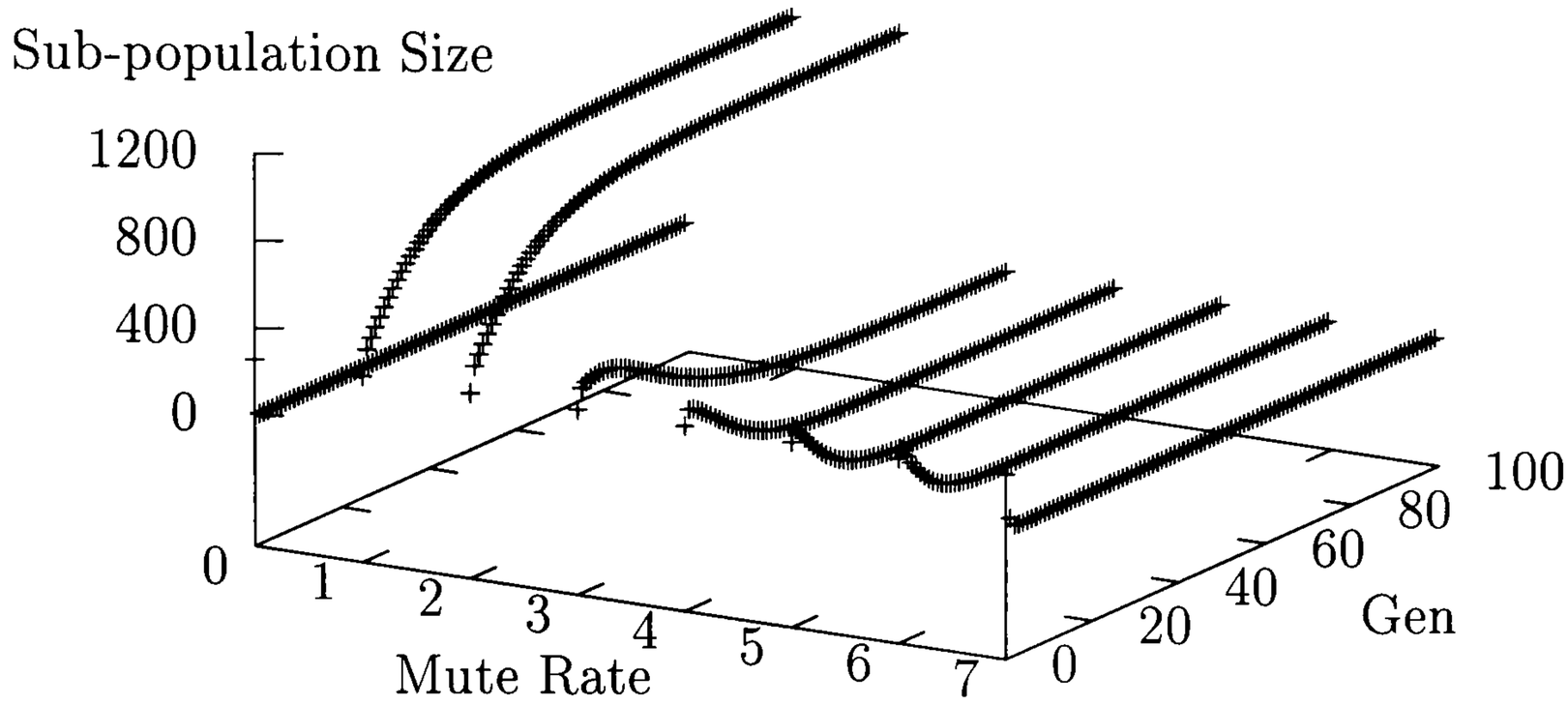


parent's mutation rate: a rate of 0 produced clones, rates from 1 to 6 produced phenotype values normally distributed around the parental phenotype with respectively increasing variance, and a rate of 7 produced offspring with uniformly random phenotype values. It should be noted that in this model the offspring inherited their parent's mutation rate – the mutation only affected the *phenotype values* of the offspring.

Each experiment was repeated with fitness functions of different variance, representing progressively more forgiving environments – with a large variance, the fitness distribution is flatter and an individual can achieve near optimal fitness, even when its phenotype is some distance from the optimum. Every generation was logically partitioned into sub-populations with equal mutation rates, and the size of each sub-population was recorded. The results from two experiments are shown in Figures 4.2 and 4.3. The source code and outputs can be found in Appendix E.

As expected, the results showed that when an environment remains constant ($k = 0$ and $f^*_{g+1} = f^*_g$), selection favours the lowest possible mutation

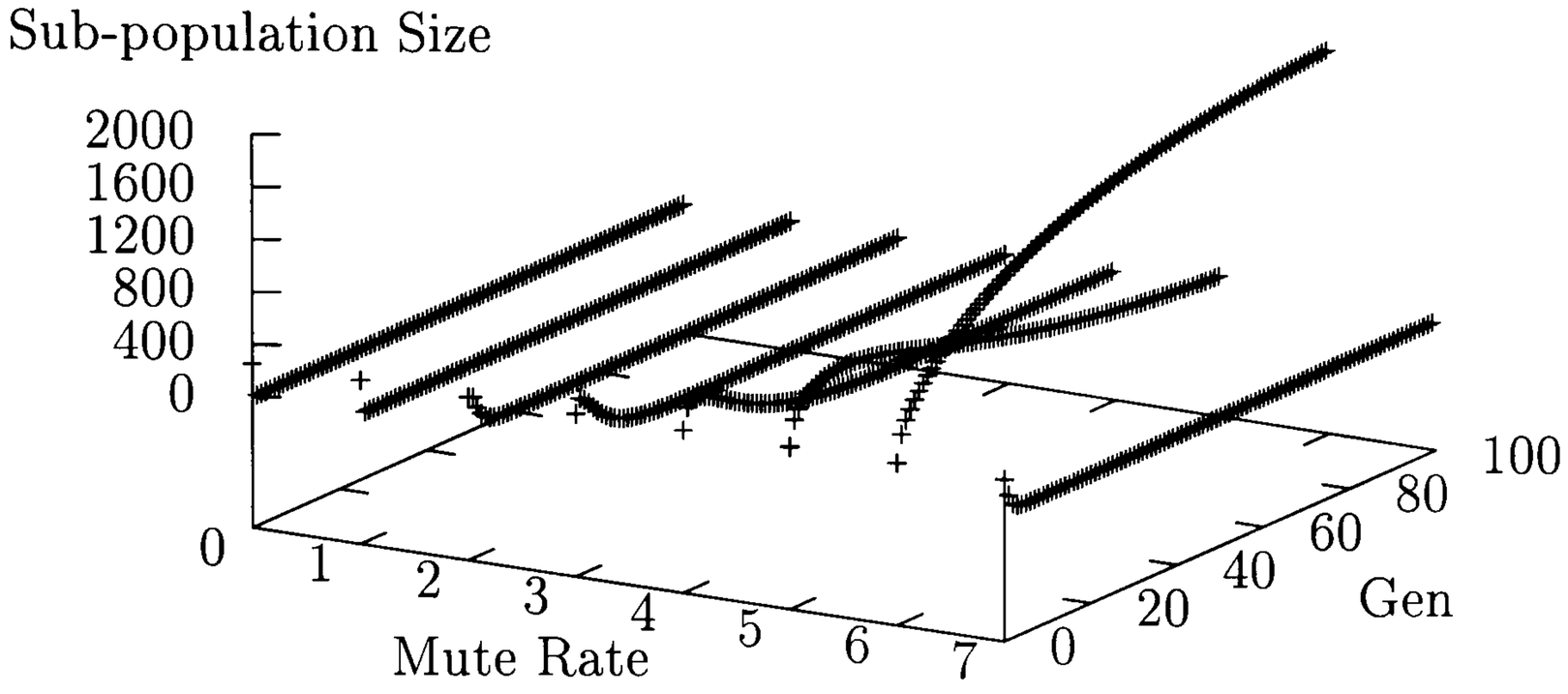
Figure 4.2: Results from simple model with 'spike' fitness (zero variance) and with the optimum phenotype moving one step per generation ($k = 1$)



rate. For dynamic environments, the optimum depended solely on the speed of environmental change – as the variance of the fitness function was increased the only effect was an increase in the number of generations it took for the optimal sub-population(s) to dominate the population. The relationship between the optimal mutation rate and the speed of environmental change is illustrated in Figure 4.4.

The experiments described above were all based on a deterministic model. Each experiment was also run stochastically a number of times, using a genetic algorithm and 'roulette wheel' selection, in order to check that stochastic effects made no difference to the results. The only significant difference was that random drift ensured that there was only ever one dominant sub-population, even though the optimum mutation rate fell exactly between two sub-populations (cf. Figure 4.2).

Figure 4.3: Results from simple model with 'spike' fitness (zero variance) and with the optimum phenotype moving two steps per generation ($k = 2$)

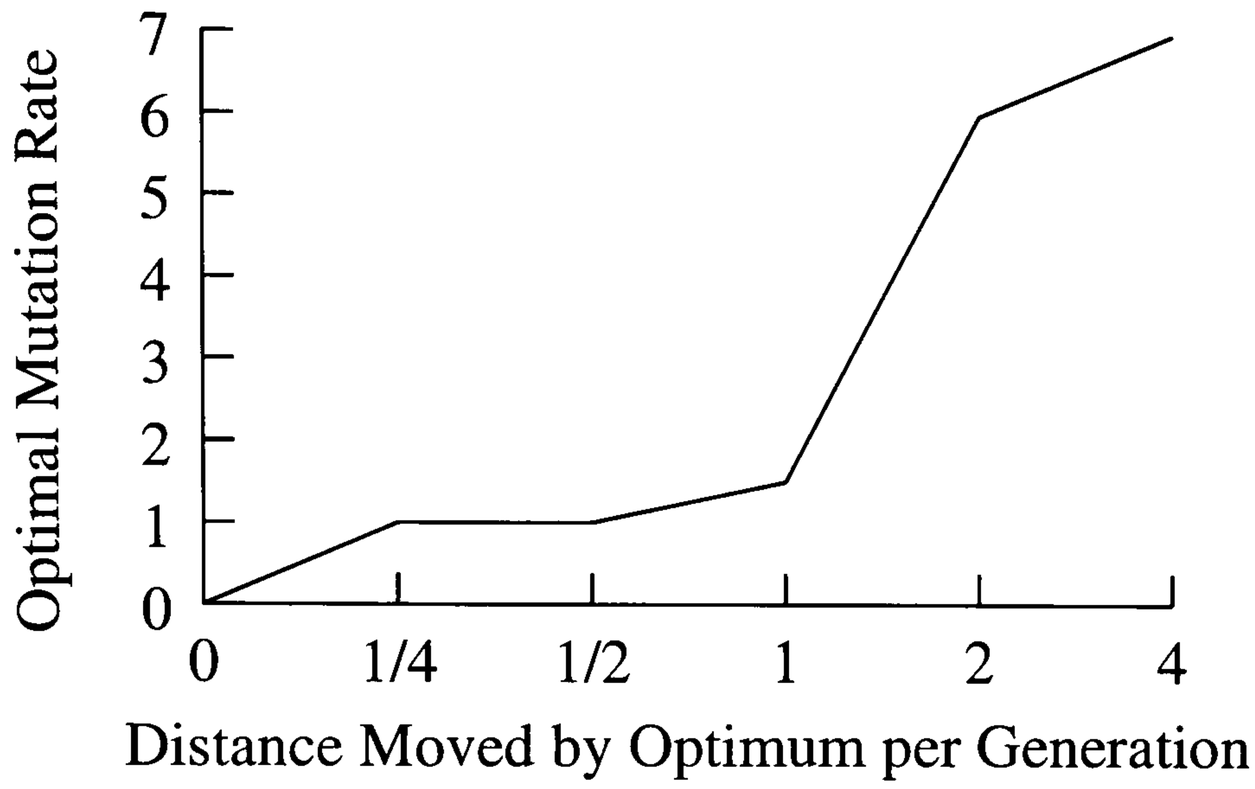


These experiments have demonstrated that there is indeed a selective advantage for increased mutation rates within dynamic environments and that hitch-hiking is not necessarily the sole explanation for the maintenance of mutator alleles within some populations.

4.1.3 A Simulation with Mutators

Although the simple model of evolving mutation rates, described in Section 4.1.2 shows that there can exist a selective pressure for increased mutation rates, the model does not include two important, complicating factors: the effect that a mutator would have upon itself, and the effects of crossover within sexual populations. Consequently, a second model was created (see Appendix F).

Figure 4.4: Graph of optimal mutation rate against speed of environmental change in simple model with 'spike' fitness (zero variance)



A standard genetic algorithm was used in this second model, each individual being represented by a fixed-length bitstring. The first bit within each individual represented its mutation rate: a 0 for a normal, low mutation rate (typically set at 0.001); and a 1 for a high mutation rate (typically set to be an order of magnitude greater than the low mutation rate). Unlike the previous model, this 'mute-boost' bit affected itself, so the probability of a mutator 1 bit mutating to an antimutator 0 bit was greater than the mutation probability in the other direction. The remaining bits in each individual represented its phenotype value.

A series of experiments was carried out on populations ranging from a hundred individuals to many thousands, with bitstring lengths ranging from eight to eighty bits. Each experiment started with a uniformly random, initial population, from which the next generation was produced using standard roulette wheel reproduction, crossover and mutation; the mutation rate being determined by an individual's mutation rate bit. All the experiments were run twice, with mutation being applied either before or after crossover. It

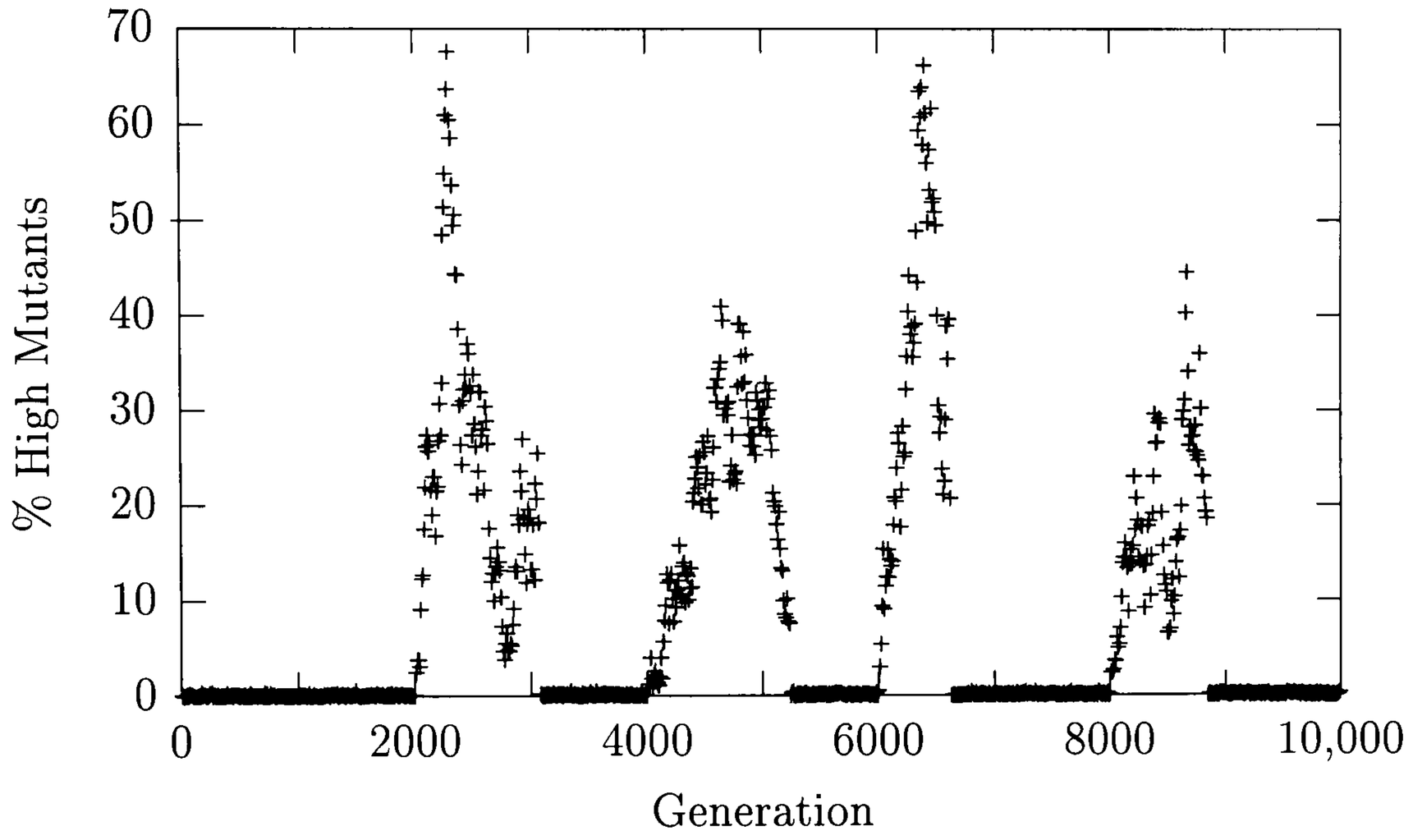
was seen that the order in which these operators were applied did not affect the results.

The experiments were designed to test the hypothesis that, when the fitness optimum moves a significant distance, the selective pressure for mutators should increase, particularly when a subset of their offspring fall within the neighbourhood of the new optimum. This is expected because of the shape of the fitness function: mutators near a tail of the fitness function will produce offspring with a greater mean fitness than will similar antimutators, due to the greater variance in the mutators' offspring phenotypes. The worst case for mutators would be a 'spike' fitness, since, unless they are close to the new optimum, their offspring will have the same fitness as the offspring of antimutators. However, random drift should ensure that the numbers of mutators in the population should rise significantly when the optimum moves, and should fall again when the new optimum is found. Also, since the variance of the phenotypes in the next generation is dependent upon the percentage of mutators within the current generation, and since a larger variance results in more phenotypes being tested, a rise in the number of mutators should reduce the time taken to find the new optimum. Consequently, the fitness function used in these experiments was a simple, oscillating, 'spike' fitness, in which the single optimum moved between two distant phenotypes once every k generations. A range of fitness values was tried, typically chosen so that the fitness optimum was an order of magnitude greater than the other fitness values.

The population was monitored for many thousand generations and, for each generation, the percentage of mutators was recorded. As expected, the experimental results showed a rise in the number of mutators when the fitness

optimum moved, and a subsequent fall when the new optimum was found. The results from a typical experiment can be seen in Figure 4.5.

Figure 4.5: Results for the simulation with mutators, with the fitness optimum changing every 2000 generations



4.1.4 Summary of Results

In a fast-changing environment an allele for increased mutation can outperform a low mutation allele in terms of reproductive success, even if it increases mutation within itself. This is because an increase in mutation increases the phenotype space sampled by the offspring of the individual carrying the high mutation allele (since the variance is greater) and consequently, in the absence of any significant selective pressure near the parent's phenotype, enlarging the sample space increases the chance of finding a new optimum, thereby raising the mean offspring fitness.

To apply this mechanism to genetic algorithms a simple modification is required. A single 'mute-boost' bit can be associated with each gene that contributes to the part of the phenotype affected by the dynamic fitness function. This bit will automatically raise the mutation rate when the selective pressure drops for that gene, and will lower it again when a new selective pressure emerges, thus improving the ability of the population to track the optima within fast-changing environments. This technique should help genetic algorithm systems keep up with fast-paced environments, allowing them to track or find the optima as they change across generations in environments in which, without this 'automutation' mechanism, they would randomly drift.

It is natural to look for possible alternatives to mutation boosting to solve the problem of tracking fast-paced environments. However, since the problem is often a blind search for the new optima, standard search techniques based on hill-climbing are ineffective. Also, since the optima continue to change, the mutation generated random search is in general as effective as any ordered, exhaustive search.

4.1.5 A New Type of Gene?

Traditional 'selfish gene' descriptions of evolutionary adaptation (Dawkins 1976, Maynard Smith & Szathmáry 1995) centre on the selective pressures that tend to favour certain subsets of alleles from the genepool, resulting in an increase in frequency of selected alleles for each locus on the chromosome(s). In other words, an allele which in some sense works well, on average, with the alleles at other loci in the current population should prosper. For this

to happen, i.e. for selective pressure to be present, the gene should affect the phenotype in some significant way. Introns are not subject to selective pressure as they do not (it is assumed) affect the phenotype, hence they are classed as 'junk DNA'.[1]

In what way does a gene for replication accuracy, a 'mutation rate' gene, affect the phenotype? It certainly doesn't affect the current generation's phenotype, and its only effect on the next generation is to alter the distribution of phenotypes[2] – a phenotypic meta-effect. Exons appear to thrive by solving problems that exist at an instant (in one particular generation). Genes such as 'mutation rate' genes – perhaps referred to as 'mexons' – appear to thrive by solving problems that only exist over an interval (a number of generations). Genes that regulate mutation rate can be thought of as not related to the phenotype of the individual, but rather to the coefficient of relatedness between the individual and its offspring. This suggests that a different form of selective pressure exists within the chromosomes of individuals in fast changing environments.

A consequence of this is a decrease in the likelihood of any advantage being conferred through kin selective cooperation in a fast changing environment, as the emergence of mutator alleles automatically lowers the coefficient of relatedness between individuals, thus increasing the criterion for advantageous kin altruism on the right hand side of Hamilton's inequality.

---

[1]Introns might be useful in a chromosome since, if they don't fall between two close genes they reduce the probability that crossover will occur between the genes, thus increasing the strength of the link between them. This suggestion has also been made by Nordin, Francone & Banzhaf (1996) with reference to genetic programs.

[2]For a unimodal fitness function and Gaussian mutation the effect of an increased mutation rate is to flatten the distribution.

4.1.6 An Alternative to Mutation

Since mutation causes problems for cooperation based on kin selection, can it be replaced by an alternative process? One possibility is the addition of an alternative way of calculating the fitness value from whatever fitness function is being used. The concept is simple. Instead of using mutation during each generation of an evolving computer simulated population, randomly divide the current population at each generation into pairs and, when a 'Hamming bit' is set in the individual, calculate the sum of the bits resulting from an XOR operation on a pair (i.e. their Hamming distance). Scale this value and use it instead of the fitness function to provide the individual's fitness value. As with an increase in the mutation rate, this method increases the population diversity, but without the detrimental effects on kin selective cooperation. Section 4.2 compares this method with changing the mutation rate.

4.2 Changing the Hamming Distance

In a stable environment, where the most successful replicators in generation k are likely to be the most successful in generation $k+1$, it would appear that a high degree of conformity both within and between generations is desirable in evolving populations. In this way, the best (fittest) phenotype yet found is maximally exploited. This can be seen in GA populations where the increase in conformity is so rapid that it often results in premature convergence. Consequently, since the best yet found is not necessarily the global optimum, if one exists, some diversity is usually provided by mutation to

allow the population to explore the fitness landscape further (the crossover operator can also help with exploring the fitness landscape when a population is itself diverse but not when it has converged). Getting the balance right between exploring and exploiting is thus a matter of setting the correct level of mutation within the population.

When the environment changes over time, Section 4.1 (also in Watson & Messer (1999)) has shown that the optimal level of mutation increases as the rate of environmental change increases. It was also shown that by including an 'automute' bit in every individual within a GA population, which boosts an individual's own mutation rate when set, the population maintained a low, mean mutation rate when the environment was stable and automatically raised the mean mutation rate when the environment changed. It was argued that this behaviour would enable an automute GA to outperform a standard GA at keeping track of fitness optima within a fast-changing environment, thus improving overall performance without any significant increase in algorithm complexity and without the need for any extra external control of the population. This section develops this self-adapting, mutation-based diversity approach by comparing the performance of both a standard GA and an automute GA within a simple model of a fast-changing environment and, more importantly, by also introducing an apparently equally effective method of controlling population diversity, based on the Hamming distance between pairs of individuals. It is argued that this 'autoham' GA is more suited to cooperative evolutionary systems since it does not rely on an increase in mutational 'noise' to provide an increase in diversity.

Before comparing the performance of an autoham GA with that of an automute GA, a comparison is made between a standard GA and an automute

GA to check that an automute GA can really outperform a standard GA at finding optima in fast-paced environments. The same program is then used to test the performance of an autoham GA.

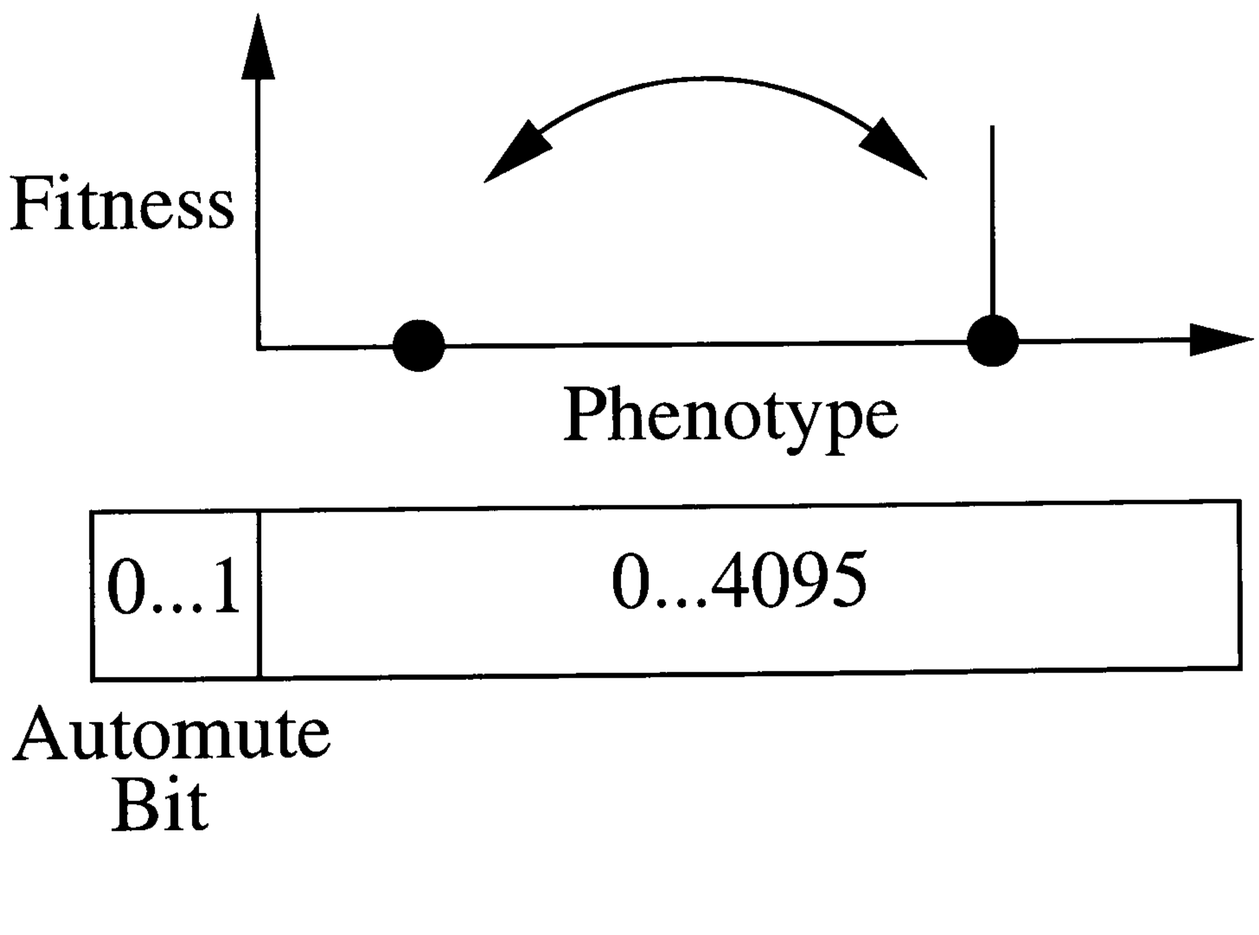
4.2.1 Mutation-Based Diversity Revisited

A few definitions are needed before describing the experiment to compare the performance of a standard GA and an automute GA. The 'standard' GA used in this paper consists of a fixed-size population of 300 individuals per generation, each individual represented by a fixed-length bitstring (13 bits). The next generation was produced using roulette-wheel selection followed by one-point crossover of 70 per cent of the paired individuals and then each bit was mutated with a probability of either 0.001 or 0.002, depending on the experimental run. In the standard GA, the rightmost 12 bits were interpreted as representing a binary integer phenotype, the value of the first bit being ignored. An automute GA is identical to the standard GA except for the interpretation of the first bit: in an automute GA, when the first bit is set for an individual it will undergo mutation at a boosted rate. In the experiment the 'muteboost' factor was $5, 10, 20$ or 50 times the base mutation rate, depending on the experimental run.

To compare the performance of a standard GA and an automute GA, the simplest possible dynamic environment was chosen, consisting of a single optimum fitness of 20, all other phenotypes producing a fitness of one. At the start of an experimental run, the initial population was allowed to evolve until 95 per cent of the individuals had converged to the optimum. Then the optimum was moved to a distant phenotype and the number of generations

taken until 95 per cent of the population had converged on the new optimum was recorded. The optimum was then swapped back to its original position and the experiment continued, recording the number of generations taken until convergence. The optimum was swapped 500 times in each experimental run and the mean number of generations taken to find the new optimum, together with the associated standard deviation, was calculated. The experimental design is summarised in Figure 4.6 and the source code and output is included in Appendix G. To cater for populations that failed to find the new optimum, an upper limit of 4000 generations was used: if a population failed to find the new optimum after 4000 generations then the optimum was swapped and the number of generations taken to find the optimum was recorded as 4000.

Figure 4.6: Chromosome format and fitness function for an automute GA in a simple, dynamic environment



The design of the experimental fitness function was influenced by the work of van Nimwegen & Crutchfield (1999) who have shown that a metastable population, i.e. one that is temporarily 'stuck' at a suboptimal fitness, will

most likely reach a higher fitness by crossing an entropy barrier (a plateau in the fitness landscape) rather than a fitness barrier (a valley in the fitness landscape). Consequently, the experimental fitness landscape was designed so that the population has to cross a base fitness plateau to find the new optimum (cf. Grefenstette's fitness landscape (Grefenstette 1999)).

The two positions for the optimum were chosen so as not to favour the automute GA. Mutation acts on a population as a spring-like force, dragging each bit position away from the extrema of all ones or all zeros towards a state of half ones and half zeros. Crossover combines individuals from one generation into the next, resulting in a mutational bias towards individuals containing half ones and half zeros. Thus, the two positions for the optimum were chosen as follows:

Optimum position A: 0100 0000 1001
Optimum position B: 1101 0110 1111

where the number of ones in position A is one quarter of the length of the phenotype and the number of ones in position B is three-quarters of the phenotype length (i.e. 3 and 9 out of 12 respectively).

The experimental results are shown in Table 4.1. These results show that an automute GA outperforms a standard GA at keeping track of the fitness optimum – the automute GA needing approximately half the number of generations than the standard GA before the new optimum is found. It would appear that by adding a single bit to each individual and by modifying the mutation operation slightly, a standard GA's ability to track dynamic environments can be improved dramatically.

Table 4.1: Number of generations needed to find new optimum fitness in a simple, dynamic environment for both a standard GA and an automute GA

		Standard GA		Automute GA	
Muterate	Muteboost	Mean	Std. Dev.	Mean	Std. Dev.
0.001	5	1163	1050	563	501
	10			528	512
	20			502	448
	50			544	473
0.002	5	468	353	288	181
	10			266	168
	20			253	165
	50			266	189

Mutation-Based Diversity and Cooperative Systems

A consequence of increasing the mutation rate of an individual is that its offspring will become less like itself. If the individual is part of a cooperating population then increasing the mutation rate will tend to make the environment more hostile towards cooperation. This is because the two main mechanisms that produce cooperating populations – kin selection and reciprocal altruism (Hamilton 1964, Maynard Smith 1982*b*) – are both detrimentally affected by an increase in mutational 'noise'.

Since GAs seem to be one of the few research areas in computer science that are suitable for developing cooperative systems in dynamic environments it would be helpful if a method for controlling diversity could be found that does not rely on increasing mutation rates. The next section outlines just such a method.

4.2.2 Frequency-Based Diversity

Although controlling diversity by altering the rate of mutation is effective, it is not the only method. Previous work has been done on self-adaptive GAs that don't alter the mutation rate, for instance, Deb & Beyer (2001), who use simulated binary crossover. But, like many of the mutation-based approaches to diversity, the emphasis of the previous research has been on avoiding premature convergence (see Rudolph (1997) and Rudolph (2001)). So, what method could be used in place of mutation-based diversity to help GAs cope with fast-paced environments?

It is well known that diversity is maintained in populations in which an individual's rarity confers a selective advantage. While it would be possible to keep statistics on the number of individuals with identical genotypes in any given population it would be too much of a computational burden. So a method of controlling diversity based on the frequency of occurrence of genes within a population, which does not require the maintenance of any global statistics, would offer a possible alternative to the mutation-based control of diversity within an automute GA. Such a method would also be preferable within cooperative systems as the population diversity would not be controlled by increasing the mutation rate. It is proposed that a modification of the fitness function to include a measure of the Hamming distance between two individuals will provide such a frequency-based control of diversity.

Controlling Diversity with an Autoham GA

Consider a change to an automute GA so that the setting of the first bit in an individual (now called the 'autoham' bit), rather than increasing the mutation rate, forces its fitness to be based on the Hamming distance between it and another randomly chosen individual from the population. As long as the maximum Hamming-based fitness is less than the environmental fitness function's optimum fitness, when the environment is stable the population would be expected to converge on the environmental optimum. Now, if the environment changes and the optimum moves, a mutated (rare) individual can gain a higher fitness if it is based on Hamming distance than if the autoham bit is not set and the environmental base fitness of one is assigned to it (assuming that it is not at the new optimum). Thus, it would be expected that the population would become dominated by autoham-on individuals until the new optimum is reached. However, the initial experimental results for this 'autoham' GA, using the same experimental design as before, were not promising.

When the optimum had been found the population did indeed converge to almost all autoham-off individuals (mutation creating a few autoham-on individuals at random). And when the optimum was moved the population became dominated by autoham-on individuals. But they failed to find the new optimum within 4000 generations. It was conjectured that this was because the population quickly evolved to a very stable state in which each individual was as far away (in terms of Hamming distance) from the others as possible. To reduce this effect it was decided to modify the interpretation of the autoham bit to mean that if it was set then there was a non-zero

probability (Hprob.) that the individual's fitness would be based on Hamming distance rather than environmental fitness. Experimental runs were performed using this autoham GA with different values for the Hprob. value – the results are presented in Table 4.2 along with the best results for both the standard and automute GAs.

Table 4.2: Number of generations needed to find new optimum fitness in a simple, dynamic environment for a standard, an automute and an autoham GA

		Standard GA		Automute GA		Autoham GA	
Muterate	Hprob.	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
0.001	0.05	1163	1050	502	448	369	224
	0.1					346	172
	0.2					253	112
	0.5					232	136
0.002	0.05	468	353	253	165	260	147
	0.1					240	108
	0.2					192	76
	0.5					212	139

The experimental results suggest that an autoham GA not only outperforms a standard GA at tracking the optimum in a fast-changing environment but that it also significantly outperforms an automute GA. Since the autoham GA does not suffer from the disadvantages associated with systems that increase the mutation rate, and since it only requires one extra bit per individual and a slight change to the calculation of fitnesses, it would seem to be preferable to both standard and automute GAs for dynamic systems.

especially for cooperative systems.

Since both automute and autoham bits improve the performance of GAs in fast-paced environments it seems natural to consider what effect might result from using both at the same time. This could be done by using two bits, or by reinterpreting a single bit, when set, to signify that the mutation rate for the individual is to be boosted and, simultaneously, that there is a probability greater than zero that it will use a fitness based on Hamming distance in place of the original fitness function.

A further modification of interest would be to include a complement of one of the members of the population in each generation, thereby ensuring that there exists both a one and a zero at every locus in the current population. This would seem to be an easy way of 'kick starting' any increase in diversity when required.

The experiments were performed again with the addition of two and one bit autoham and automute combined, and with the inclusion of a complementary member for each generation. The results are summarised in Table 4.3. Appendix G contains both the source code and output.

It would appear that the best combination is a single bit that controls both automutation and autohamming with the addition of a complement.

4.2.3 Summary of Results

It has been shown that an automute, an autoham GA and a combined automute and autoham GA can significantly outperform a standard GA at

Table 4.3: Comparison of the number of generations needed to find new optimum fitness in a simple, dynamic environment for GA systems with and without automute and autoham control bit(s) and complements

	No Complement		Complement	
GA System	Mean	Std. Dev.	Mean	Std. Dev.
Standard	484	407	188	112
Automute	238	143	137	72
Autoham	183	77	148	79
Automute & Autoham, two control bits	134	51	138	70
Automute & Autoham, one control bit	81	46	71	45

keeping track of the fitness optimum in a fast-changing environment. The autoham GA also benefits from not relying on an increase of the mutation rate to control population diversity, thereby making it more suitable for cooperative systems. Since neither technique requires the maintenance of any global statistics and since the extra computational load is minimal both techniques should be an improvement over standard GAs in dynamic environments.

By including a complementary member into each generation the performance is further improved, again without any significant increase in computational effort.

Although the emphasis has been on introducing a single automute or autoham bit into each individual there is no reason why such a bit could not be attached to each gene in the chromosome. This bit would automatically raise the mutation rate or the Hamming fitness probability when the selective

pressure drops for that gene, and would lower it again when a new selective pressure emerges, thus producing gene-level control of the diversity within the population. However, while posing no problems for mutation, the way in which an autoham bit for a part of a chromosome would affect an individual's fitness is more difficult to define. It would probably require the use of a real-valued parameter that starts low and slowly increases across generations – an added complication that detracts from the simplicity of a global autoham bit.

Lastly, whenever the relative performances of search algorithms are discussed, a mention should be made of the 'No Free Lunch' theorem (Wolpert & Macready 1997). In their important paper, Wolpert & Macready state that,

> 'Roughly speaking, we show that for both static and time-dependent optimization problems, the average performance of any pair of [search] algorithms across all possible problems is identical.'

Nevertheless, when the problem domain is restricted to fast-paced environments, the modified GA systems described above do offer significantly better performance than a standard GA.

Chapter 5

Stabilising Kin Selection

It has been seen that both kin instability and fast-paced environments pose problems for cooperative systems. Chapter 4 has shown how the problem of fast-paced environments can be overcome. So how might kin selection be stabilised?

In spite of the inherent instability of kin selection, stabilising it would seem to be straightforward in GA systems. Since the genotype is easily accessed for each individual, the destructive effects of kin mimics could be removed by identifying the bits associated with altruism and only allowing individuals to cooperate if their altruism bits match. However, in many systems the bits cannot be identified. For example, when GAs are used to evolve weights in artificial neural networks a sequence of bits determines the weight of a link, quite how it affects the behaviour of an artificial neural network is not apparent.

Perhaps, by restricting cooperation to interactions between clones, co-operation could emerge and be maintained? This might work in a system that is not evolving but, in evolutionary systems, adaptation relies on the

differences produced by mutation and crossover. Any such restriction would destroy adaptation amongst cooperators, yet it is precisely the adaptation that is the attraction of evolutionary cooperative systems – the ability to evolve well-adapted cooperative solutions. What is needed is a way of stabilising kin selection without having to make kin mimicry impossible. But, before attempting to stabilise kin selection, it will be instructive to consider, in more detail, why it is unstable.

5.1 An Analysis of Kin Selective Instability

Consider a population large enough to be approximated by an infinite population. The population consists of two-bit strings where the first bit is a green beard gene and the second bit is a cooperate with green beards gene (for both genes, 1 = true). Note that 00 doesn't take part in any cooperation (a selfish genotype), 11 is a kin altruist and 10 is a kin mimic (a cheater). Let the proportions of the four different genotypes in the population be α, β, γ and δ for $00, 01, 10$ and 11 respectively ($\alpha+\beta+\gamma+\delta = 1$), and let $\alpha > 0$ and $\delta > 0$. Then, the fitness function[1] for individual a when interacting with b is

$$\text{fit}_a = 2 - (a = \#1 \wedge b = 1\#) + \text{benefit}(a = 1\# \wedge b = \#1),$$

where # represents either a 0 or a 1.

[1]This function represents a two-way interaction between a pair of individuals (a, b), equivalent to two interactions from the program in Section 3.2. However, in an infinite population that is randomly paired there will be as many (a, b) pairs as there are (b, a) pairs and so the fitness functions used here and previously are comparable.

The mean fitnesses for each of the genotypes are

$$\begin{aligned}
\text{fit}_{00} &= 2, \\
\text{fit}_{01} &= 2 - (\gamma + \delta), \\
\text{fit}_{10} &= 2 + \text{benefit}(\beta + \delta), \\
\text{fit}_{11} &= 2 - (\gamma + \delta) + \text{benefit}(\beta + \delta).
\end{aligned}$$

If $\gamma > 0$ then, for all benefits ≥ 0

$$\begin{aligned}
\text{fit}_{10} &\geq \text{fit}_{00}, \\
\text{fit}_{10} &> \text{fit}_{01}, \\
\text{fit}_{10} &> \text{fit}_{11}.
\end{aligned}$$

If $\gamma = 0$ then

$$\text{fit}_{11} > \text{fit}_{00} \quad \text{if} \quad \text{benefit}(\beta + \delta) > \delta.$$

In other words, if kin mimics exist they will wipe out any cooperation and selfishness will become stable in the population. If kin mimics don't exist and if crossover or mutation don't produce them then stable cooperation can spread throughout the population, provided that the benefit outweighs the cost. These results are summarised in Table 5.1.

A point worth mentioning with regard to the expected stable population state for a population with no possibility of kin mimics and with benefit = 1 is that if $\beta > 0$ then the kin altruists will be fitter than the selfish individuals until the value of β becomes arbitrarily close to 0, when the population dynamics become arbitrarily close to a random walk. But, since, at this point, there are likely to be more kin altruists than selfish individuals the most likely stable population state would be expected to be cooperative.

Table 5.1: Expected stable population states for three environments with different levels of benefit

	Kin Mimics	No Kin Mimics
benefit = 0	Selfish	Selfish
benefit = 1	Selfish	—
benefit = 2	Selfish	Cooperative

This doesn't alter the fact that, in the presence of kin mimics, kin selection will never produce cooperation. So, for kin selection to always encourage cooperation when it is worthwhile, the mean fitnesses will have to be altered in some way.

After consideration of the possible ways in which the fitnesses awarded for an interaction between individuals could be modified, one aspect of the interactions seemed to be significant: the asymmetry of the interactions. The next section considers the effect of asymmetric interactions on cooperation.

5.2 A Stable Solution

Table 5.2 shows the payoffs for the 'Green Beard' experiment in Section 3.2 when benefit = 2, assuming that the pair interact, swap positions and then interact again.

The asymmetries occur when individuals get different payoffs for the same interaction. If they always got the same then the matrix would be symmetric about its leading diagonal (since your partner's payoff matrix is a mirror of

Table 5.2: Payoff matrix for 'Green Beard' experiment with benefit = 2, assuming that the pair interact, swap positions and then interact again

	He's 00	He's 01	He's 10	He's 11
I'm 00	4	4	4	4
I'm 01	4	4	3	3
I'm 10	4	5	4	5
I'm 11	4	5	3	4

your own in the leading diagonal). In such a symmetric matrix it is impossible to cheat. Whenever an individual successfully increases a payoff in his row, he automatically increases the payoff received by his partner. Although, in the original experiment, only one interaction occurred, in a large enough population, for every interaction (a, b) there is a corresponding (b, a).

As a result of considering interaction symmetry it was proposed that, by combining the fitnesses from both partners in an interaction and then sharing the fitness equally between the interacting individuals, kin selection could be stabilised.

Unfortunately, the obvious term for such an operation – fitness sharing – is already used to refer to something quite different. Sareni & Krähenbühl (1998, p. 97) explain that it was originally proposed by Holland (1975) and developed by Goldberg & Richardson (1987). Fitness sharing is a technique that modifies the search landscape by reducing the payoff in densely populated regions. As such, it is categorised as a niching method and is useful for evolutionary computing systems that include multimodal fitness functions

(cf. Li, Balazs, Parks & Clarkson (2002)). Sareni & Krähenbühl also state that it is probably the best known and used niching technique to reduce the effect of genetic drift in standard GAs and thus to promote the formation of stable subpopulations in the neighbourhood of optimal solutions. As a result, a different term had to be used and the method proposed in this work is referred to as 'group fitness'.

By repeating the previous method of analysis with the addition of group fitness, the mean fitnesses for each of the genotypes are

$$\begin{aligned}
\text{fit}_{00} &= 2, \\
\text{fit}_{01} &= 2(\alpha+\beta) + (\gamma+\delta)(3+\text{benefit})/2, \\
\text{fit}_{10} &= 2(\alpha+\gamma) + (\beta+\delta)(3+\text{benefit})/2, \\
\text{fit}_{11} &= 2\alpha + (\beta+\gamma)(3+\text{benefit})/2 + \delta(1+\text{benefit}).
\end{aligned}$$

The expected, stable population states for environments with different levels of benefit are summarised in Table 5.3.

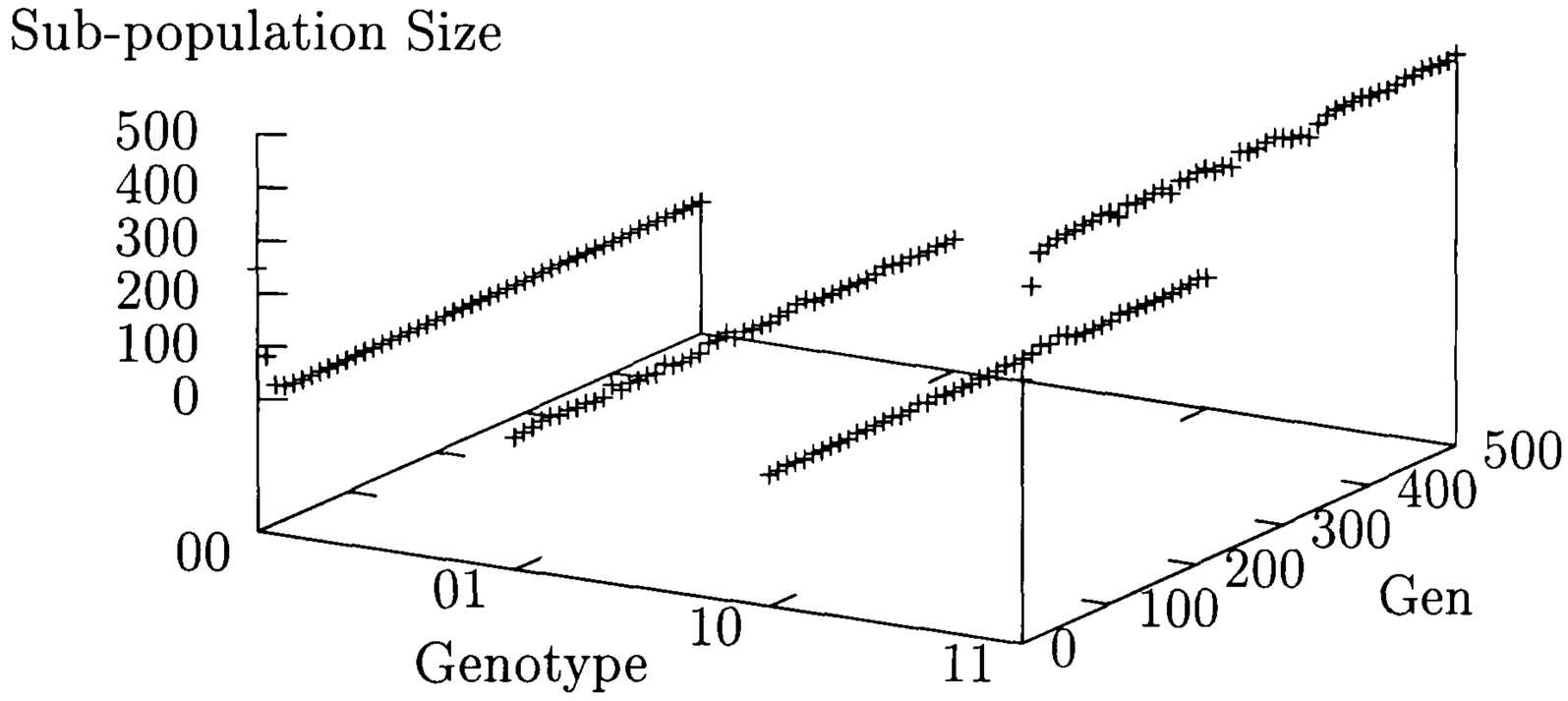
Table 5.3: Expected stable population states for three environments that employ group fitness with different levels of benefit

	Kin Mimics	No Kin Mimics
benefit = 0	Selfish	Selfish
benefit = 1	—	—
benefit = 2	Cooperative	Cooperative

The instability of kin selection, caused by kin mimics, has disappeared. With group fitness, if an environment is such that it pays to be cooperative then cooperation will flourish.

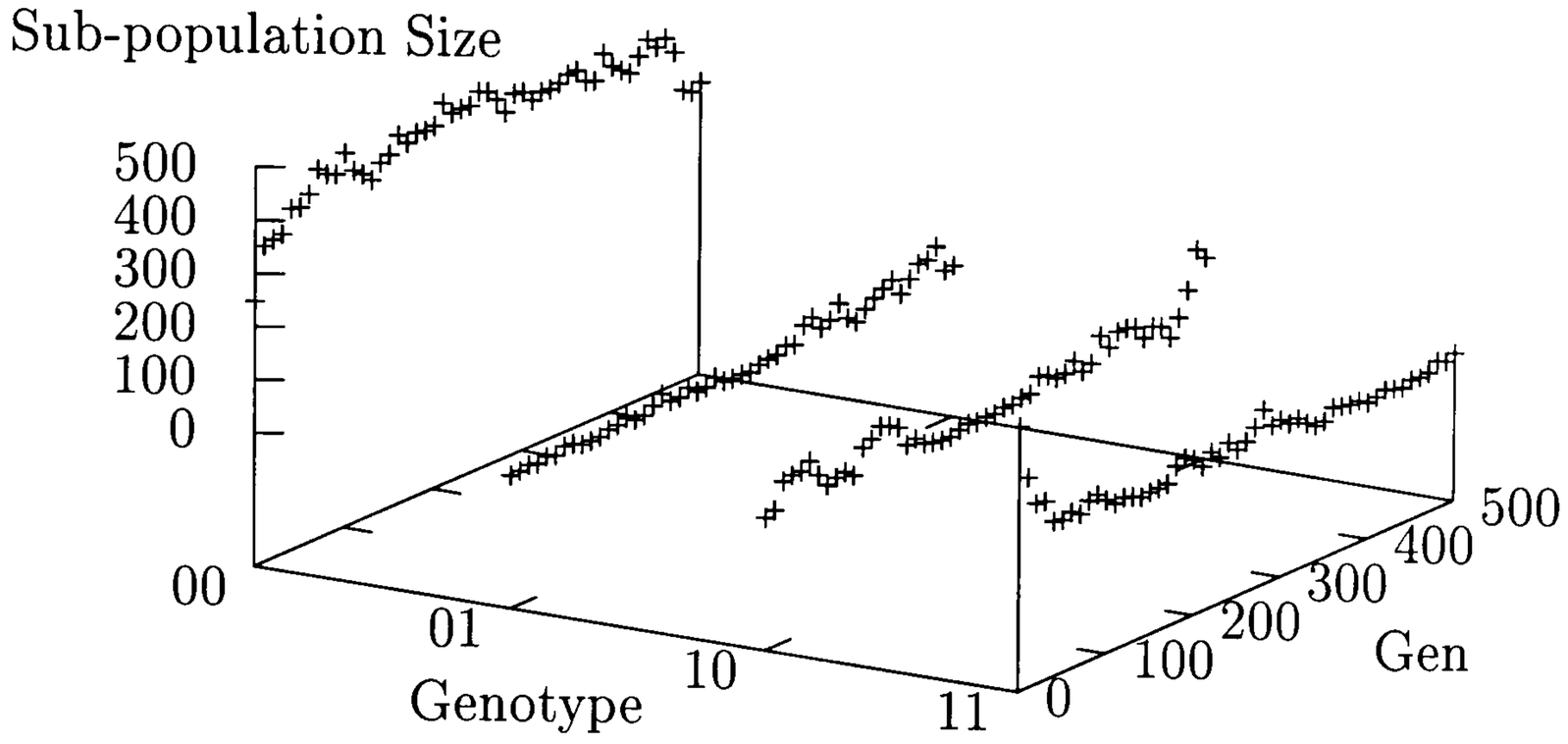
To test group fitness, a modified version of the original kin selection program was produced in which the fitnesses for each pair were combined and then shared out equally between the two interacting individuals. The source code and output is included in Appendix B and the results can be seen in Figures 5.1, 5.2 and 5.3.

Figure 5.1: Results for group fitness 'Green Beard' effect program with no crossover



The results confirm the theory. In environments in which cooperation is beneficial, kin selection produces a stable population of cooperating individuals (11s in the Figures). The presence of kin mimics (10s), or the use of mutation and crossover that might produce them, does not affect the stability of the cooperating population. Group fitness stabilises kin selection.

Figure 5.2: Results for group fitness 'Green Beard' effect program with no crossover and benefit= 0

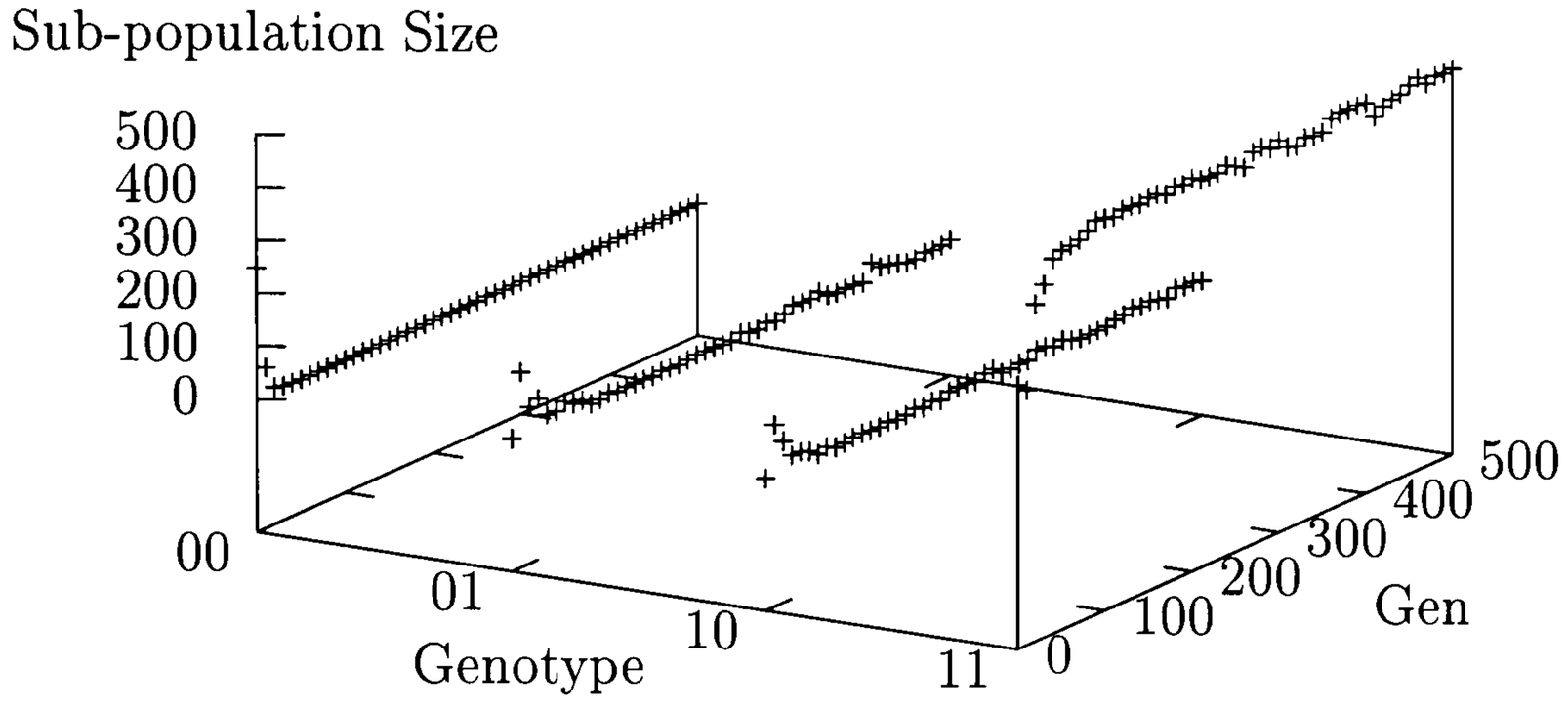


5.3 Discussion

It might seem that group fitness is simply another form of group selectionism (Wynne-Edwards 1962). In the same way that group selectionism treats a population as though it is a super-organism, the effect of group fitness would appear to be to join two individuals into one super-individual. However, this is not the case. Group fitness is not restricted to systems in which individuals 'pair for life' but is equally applicable to a system in which an individual is involved in many interactions during its lifetime with a variety of partners. While a single interaction might be considered a super-organism, the collection of individuals that interact in a generation will not, in general, all receive the same fitness.

Group fitness need not only be applied to interacting pairs. If cooperation amongst larger groups is required then group fitness is equally applicable.

Figure 5.3: Results for group fitness 'Green Beard' effect program with both crossover and mutation

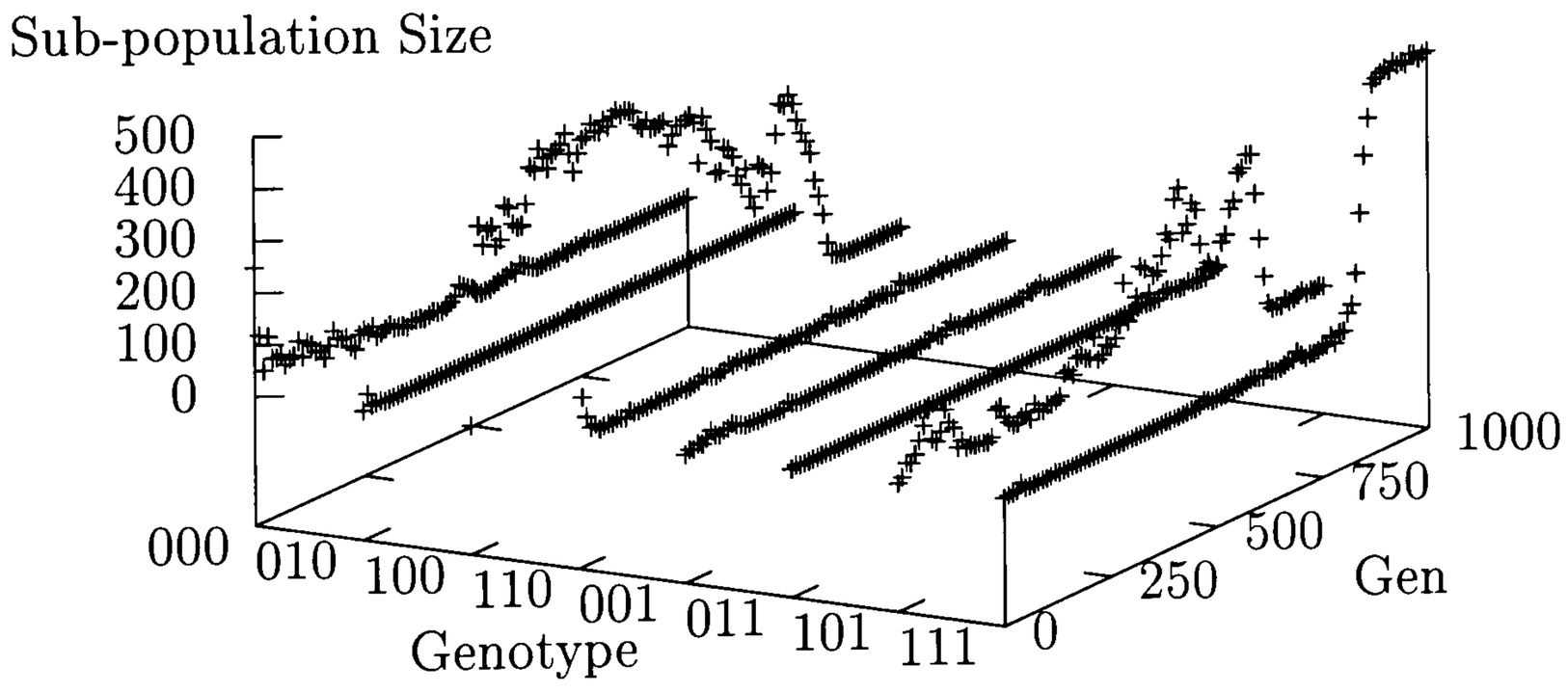


During its lifetime, an individual could conceivably interact in pairs, triples and in even larger groups, its fitness being the combined fitness from all the interactions. The effect will be the same: if cooperation is beneficial then cooperation will emerge.

For biologists, the discovery that kin selection is not a sufficient explanation for the maintenance of cooperation may come as a blow. While many of the existing explanations of cooperative behaviour are based upon reciprocal altruism, and are thus unaffected, not all are. Yet, perhaps for the first time, it is now possible to give a convincing explanation for the cooperation that exists between groups of genes in the same chromosome or individual, or the cooperation between mitochondria and their host cell, or why parents cooperate to bring up their offspring: it's because they all get the same fitness so non-cooperation would only be cheating themselves.

One last point. Maynard Smith (1958, p. 199) states that an explanation is needed for the emergence of cooperative populations. In the work presented so far on group fitness, the mechanism was built into the environment. For GA systems in which cooperation is required this is the recommended approach. However, a modified version of the original kin selection program has been produced in which group fitness is not built in. Instead, an extra bit is appended to each individual to represent whether or not it wishes to use group fitness. The source code and output are included in Appendix C and the results can be seen in Figure 5.4.

Figure 5.4: Results for 'Green Beard' Effect program with an extra group fitness bit



The initial population does not contain any group fitness advocates (the third bit of each individual in the initial population is zero) but, as Figure 5.4 shows, the population eventually becomes dominated by cooperating individuals using group fitness (111s). The output for 9999 generations in Appendix C.3 shows that the population is stable.

It would appear that a willingness to use group fitness can emerge and that, as a consequence, stable, cooperative populations can develop. It is expected that group fitness can be used with automute and autoham bits (and with complements) to reduce the number of generations needed for cooperation to emerge. Demonstrating this is a topic for further research.

Chapter 6

Conclusions

The investigation into cooperative behaviour has produced the following conclusions:

- The standard explanation for the emergence and persistence of cooperation within evolving populations – kin selection – is not a sufficient explanation as kin selective cooperation is destroyed by kin mimics.

- Another problem for cooperation is the effect of a fast-paced environment. For cooperation to emerge there needs to be enough time for selection, mutation and crossover to produce a subpopulation of cooperating individuals, and, in a fast-paced environment, there may well not be enough time for this to happen. Since the other members of a generation are part of an individual's environment, evolving systems are naturally fast-paced.

- The length of time needed for cooperation to emerge can be reduced through the use of automute and/or autoham bits, and the addition of a complementary member of each generation. These mechanisms allow an evolving GA population to cope with fast-paced environments

without the need for any global statistics to be kept and with no significant computational overhead. A further advantage of autoham is that it doesn't need to lower the coefficient of relatedness between parents and their offspring, an important consideration for cooperative systems based on kin selection.

- Stable cooperative populations can be created and maintained by using group fitness. The population is grouped, the members of each group interact and their resulting fitnesses are collected and shared equally amongst all group members. With no significant computational overhead and without the need for global statistics, cooperation can emerge whenever it is advantageous. Kin mimics do not cause any problems and selective pressure removes them from the population.

This research has also highlighted the need for further work to develop and extend our understanding of cooperative systems. Section 1.1 includes descriptions of research topics that could provide greater understanding, most notably a study of the effect of environmental topology on the emergence of cooperation. Chapter 4 also provides the basis for further study: investigating the characteristics of automute and autoham with different classes of fitness landscapes; confirming the ability of automute bits in particular to be useful when associated with several genes on a chromosome; and testing the effectiveness of automute, autoham and/or complementary members when combined with group fitness. It is also hoped that these techniques will shortly be applied to a real-world problem to determine whether the theoretical benefits actually enable an evolutionary system to evolve effective, cooperative solutions. Early experimental results in this area are promising.

In Section 1.1 it was stated that this research could be defined to be a search for the answer to the question:

> 'If kin selection is inherently unstable, where is the stabilising factor; and what alternative mechanisms can be used to encourage cooperation to evolve in computational environments?'

The answer is:

> 'Kin selective cooperation is demonstrably unstable.The stabilising factor for kin selection is group fitness. There are two mechanisms that can also encourage cooperation to occur: automute and autoham. Autoham might be preferable as it doesn't adversely affect the coefficient of relatedness between parents and their offspring but automute can more easily be used to allow each gene to alter its own mutation rate. The performance of both automute and autoham can be further improved by using a complementary member in each generation.'

Appendix A

'Green Beard' Effect Program: kin1

A.1 Source Code for kin1.c

```
/*
PROGRAM:  kin1.c
AUTHOR:   Tim Watson
MODIFIED: April 2003

Test to determine the properties of a GA-like population
with kin selective altruism ('Green Beard' Effect).

  o Two-bit strings (beard colour, altruist)
  o Fixed probability that crossover will dislocate bits
  o Static population size (all generations equal size)
  o Proportion of 00, 01, 10, 11 in initial population fixed
  o Standard proportionate reproduction
  o One-point crossover
  o Statistics - number of 00s 01s, 10s and 11s in each gen

Example: kin1 -p500 -g500 -c50 -i2 -b3 -m500 -r3
popsize=500,
generations=500,
crossover prob=50%,
initial population=type 2,
benefit=3,
muterate=1/500,
```

```
          random seed=3.
*/


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "r250.h"
#include "randlcg.h"


#define REPS        10000     /* used by pairs */
#define POPSIZE       500     /* must be even */
#define POPMAX       2000
#define CROSSOVER      50
#define INITPOP         1
#define BENEFIT         2
#define GENERATIONS  1000
#define MUTERATE     1000     /* 0 = no mutation */
#define RANDOMIZE       2


typedef struct member *Memptr;

typedef struct member {
  char    chrom[2];
  double  value;
  Memptr  lower,
          higher;
} Member;

Member curr[POPMAX], mate[POPMAX], next[POPMAX];

int counts[4];

/* precalced prob of n intact pairs after shuffling */
int pairs[12];
```

```
void init_pop(int, int, int);
void calc_fit(int, int);
void fitness(int, int);
void add_to_tree(Member *, Member *);
void update_stats(int);
void display_stats(int);
int prod_mate(Member *, int);
void copy_member(Member *, Member *);
void next_gen(int, int, int, int);



main(int argc, char *argv[])
{
  int i, inc, popsize, gens, matesize, muterate, cross,\
      ipop, random, benefit;

  popsize = POPSIZE;
  matesize = POPSIZE;
  gens = GENERATIONS;
  cross = CROSSOVER;
  ipop = INITPOP;
  benefit = BENEFIT;
  muterate = MUTERATE;
  random = RANDOMIZE;
  inc = 1;

  while (--argc > 0 && (*++argv)[0] == '-')
    switch (*++argv[0]) {
    case 'p': if (*++argv[0])
                popsize = atoi(argv[0]);
              else if (--argc > 0)
                popsize = atoi(*++argv);
              break;
  case 'g': if (*++argv[0])
                gens = atoi(argv[0]);
              else if (--argc > 0)
                gens = atoi(*++argv);
              break;
```

```
case 's': if (*++argv[0])
            inc = atoi(argv[0]);
          else if (--argc > 0)
            inc = atoi(*++argv);
          break;
case 'c': if (*++argv[0])
            cross = atoi(argv[0]);
          else if (--argc > 0)
            cross = atoi(*++argv);
          break;
case 'i': if (*++argv[0])
            ipop = atoi(argv[0]);
          else if (--argc > 0)
            ipop = atoi(*++argv);
          break;
case 'b': if (*++argv[0])
            benefit = atoi(argv[0]);
          else if (--argc > 0)
            benefit = atoi(*++argv);
          break;
case 'M': if (*++argv[0])
            matesize = atoi(argv[0]);
          else if (--argc > 0)
            matesize = atoi(*++argv);
          break;
case 'm': if (*++argv[0])
            muterate = atoi(argv[0]);
          else if (--argc > 0)
            muterate = atoi(*++argv);
          break;
case 'r': if (*++argv[0])
            random = atoi(argv[0]);
          else if (--argc > 0)
            random = atoi(*++argv);
          break;
  default:  break;
  }

if (popsize > 0) {
  if (popsize > POPMAX)
    popsize = POPMAX;
```

```
} else
  popsize = POPSIZE;

printf("popsize=%d ipop=%d benefit=%d mute=%d cross=%d\n",
       popsize, ipop, benefit, muterate, cross);

init_pop(popsize, random, ipop);

for (i=1; i<gens; i++) {
  calc_fit(popsize, benefit);
  if ((i/inc*inc==i) || (i==1))
    display_stats(i);
  prod_mate(curr, matesize);
  next_gen(matesize, popsize, muterate, cross);
}

calc_fit(popsize, benefit);
if (i/inc*inc==i)
  display_stats(i);
  return 0;
}



/* init_pop: create initial, random population */
void init_pop(int popsize, int random, int ipop)
{
  int i, j, halfpop, sum;
  double total;

  r250_init(random);

  /* precalculate how often i intact pairs will remain
     together after random shuffling, REPS times */
halfpop = popsize>>1;
pairs[0] = REPS>>1;
for (i=1; i<12; i++) {
  total = (double)REPS;
  for (j=0; j<i; j++)
    total *= (double)(halfpop-j) / \
```

```
                                        (double)((popsize-j)*(j+1));
  total *= ((double)halfpop/ \
           (double)(popsize-i)+(double)i) / (double)(i+1);
  pairs[i] = (int)(total+0.5);
}
sum = -pairs[0];
for (i=1; i<12; i++)
  sum += pairs[i];
pairs[0] -= sum;
for (i=1; i<12; i++)
  pairs[i] += pairs[i-1];

for (i=0; i<popsize; i++)
     switch (ipop) {
     case 1:  curr[i].chrom[0] = curr[i].chrom[1] = \
                                      '0' + r250() % 2;
              break;
  case 2:  curr[i].chrom[0] = '0' + r250() % 2;
           curr[i].chrom[1] = '0' + r250() % 2;
           break;
     default: break;
     }
}



void calc_fit(int popsize, int benefit)
{
  int i;

  for (i=0; i<4; i++)
    counts[i] = 0;

  fitness(0, benefit);
  update_stats(0);
  curr[0].lower = curr[0].higher = NULL;
  curr[1].lower = curr[1].higher = NULL;
  add_to_tree(curr, curr+1);

  for (i=2; i<popsize-1; i+=2) {
    fitness(i, benefit);
```

```
  update_stats(i);
  curr[i].lower = curr[i].higher = NULL;
  curr[i+1].lower = curr[i+1].higher = NULL;
  add_to_tree(curr, curr+i);
  add_to_tree(curr, curr+i+1);
}
```

```
/* fitness: return fitness for a pair */
void fitness(int i, int benefit)
{
  if ((curr[i].chrom[1]=='1')&&(curr[i+1].chrom[0]=='1')) {
    curr[i].value = 1;
    curr[i+1].value = 2 + benefit;
  } else {
    curr[i].value = 2;
    curr[i+1].value = 2;
  }
}
```

```
/* update_stats: update the counts */
void update_stats(int i)
{
  int count;

  count = (curr[i].chrom[0]-'0'<<1) + curr[i].chrom[1]-'0';
  counts[count]++;
  i++;
  count = (curr[i].chrom[0]-'0'<<1) + curr[i].chrom[1]-'0';
  counts[count]++;
}
```

```
/* add_to_tree: add <node> to <tree>, ordered by
   <node->value> */
void add_to_tree(Member *tree, Member *node)
```

```
{
  if (node->value > tree->value)
    if (tree->higher != NULL)
      add_to_tree(tree->higher, node);
    else
      tree->higher = node;
  else if (tree->lower != NULL)
      add_to_tree(tree->lower, node);
  else
      tree->lower = node;
}



/* display_stats: display the count of each of the four
   types of individual for the current generation */
void display_stats(int gen)
{
  printf("gen%4d       00=%3d   01=%3d   10=%3d   11=%3d\n",
         gen, counts[0], counts[1], counts[2], counts[3]);
}



/* prod_mate: recursively copy best <size> nodes from
   <curr> into <mate> (in reverse order) */
int prod_mate(Member *tree, int size)
{
  if (tree != NULL) {
    if (tree->higher != NULL)
      size = prod_mate(tree->higher, size);
    if (size > 0) {
      copy_member(mate+size-1, tree);
      size = prod_mate(tree->lower, size -= 1);
    }
  }

  return size;
}
```

```
/* copy_member: copy member from <curr> to <mate>
   (destroys tree) */
void copy_member(Member *to, Member *from)
{
  strcpy(to->chrom, from->chrom);
  to->value = from->value;
}



/* next_gen: reproduce, crossover and mutate
   from <mate> to <curr> via <next> */
void next_gen(int size, int popsize, int mute, int cross)
{
  int i, j, n;
  static double choice;
  static char temp;

  for (i=1; i<size; i++)      /* calculate proportions */
    mate[i].value += mate[i-1].value;

for (i=0; i<popsize; i++) {               /* reproduce */
  choice = r250() % (int) mate[size-1].value + 1;
  j = 0;
  while (j < size-1 && choice > mate[j].value)
    j++;
  strcpy((next+i)->chrom, (mate+j)->chrom);
}

for (i=0; i<popsize; i+=2)              /* crossover */
  if (r250()%100 < cross) {
    temp = next[i].chrom[0];
    next[i].chrom[0] = next[i+1].chrom[0];
    next[i+1].chrom[0] = temp;
  }

if (mute != 0)
  for (i=0; i<popsize; i++)                    /* mutation */
    for (j=0; j<2; j++)
      if ((r250() % mute) == 0)
```

```
            next[i].chrom[j] = '0' + '1' - next[i].chrom[j];

    /* simulate random shuffling from next back to
       current generation */
    n = r250() % REPS + 1;
    for (i=0; n>pairs[i>>1]; i+=2) {
      strcpy((curr+i)->chrom, (next+i)->chrom);
      strcpy((curr+i+1)->chrom, (next+i+1)->chrom);
    }
    strcpy((curr+popsize-1)->chrom, (next+i)->chrom);
    for (i++; i<popsize-1; i+=2) {
      strcpy((curr+i-1)->chrom, (next+i)->chrom);
      strcpy((curr+i)->chrom, (next+i+1)->chrom);
    }
  }
```

A.2 Output for kin1.c: No Crossover and no Mutation

```
popsize=500 ipop=1 benefit=2 mute=0 cross=0
gen   1       00=248   01=  0   10=  0   11=252
gen  10       00= 91   01=  0   10=  0   11=409
gen  20       00= 17   01=  0   10=  0   11=483
gen  30       00=  0   01=  0   10=  0   11=500
gen  40       00=  0   01=  0   10=  0   11=500
gen  50       00=  0   01=  0   10=  0   11=500
gen  60       00=  0   01=  0   10=  0   11=500
gen  70       00=  0   01=  0   10=  0   11=500
gen  80       00=  0   01=  0   10=  0   11=500
gen  90       00=  0   01=  0   10=  0   11=500
gen 100       00=  0   01=  0   10=  0   11=500
gen 110       00=  0   01=  0   10=  0   11=500
gen 120       00=  0   01=  0   10=  0   11=500
gen 130       00=  0   01=  0   10=  0   11=500
gen 140       00=  0   01=  0   10=  0   11=500
gen 150       00=  0   01=  0   10=  0   11=500
gen 160       00=  0   01=  0   10=  0   11=500
gen 170       00=  0   01=  0   10=  0   11=500
gen 180       00=  0   01=  0   10=  0   11=500
gen 190       00=  0   01=  0   10=  0   11=500
gen 200       00=  0   01=  0   10=  0   11=500
gen 210       00=  0   01=  0   10=  0   11=500
gen 220       00=  0   01=  0   10=  0   11=500
gen 230       00=  0   01=  0   10=  0   11=500
gen 240       00=  0   01=  0   10=  0   11=500
gen 250       00=  0   01=  0   10=  0   11=500
gen 260       00=  0   01=  0   10=  0   11=500
gen 270       00=  0   01=  0   10=  0   11=500
gen 280       00=  0   01=  0   10=  0   11=500
gen 290       00=  0   01=  0   10=  0   11=500
gen 300       00=  0   01=  0   10=  0   11=500
gen 310       00=  0   01=  0   10=  0   11=500
gen 320       00=  0   01=  0   10=  0   11=500
gen 330       00=  0   01=  0   10=  0   11=500
gen 340       00=  0   01=  0   10=  0   11=500
gen 350       00=  0   01=  0   10=  0   11=500
```

gen 360	00=	0	01=	0	10=	0	11=500
gen 370	00=	0	01=	0	10=	0	11=500
gen 380	00=	0	01=	0	10=	0	11=500
gen 390	00=	0	01=	0	10=	0	11=500
gen 400	00=	0	01=	0	10=	0	11=500
gen 410	00=	0	01=	0	10=	0	11=500
gen 420	00=	0	01=	0	10=	0	11=500
gen 430	00=	0	01=	0	10=	0	11=500
gen 440	00=	0	01=	0	10=	0	11=500
gen 450	00=	0	01=	0	10=	0	11=500
gen 460	00=	0	01=	0	10=	0	11=500
gen 470	00=	0	01=	0	10=	0	11=500
gen 480	00=	0	01=	0	10=	0	11=500
gen 490	00=	0	01=	0	10=	0	11=500
gen 500	00=	0	01=	0	10=	0	11=500

A.3 Output for kin1.c: No Crossover, Mutation or Benefit

```
popsize=500 ipop=1 benefit=0 mute=0 cross=0
gen   1       00=248   01=  0   10=  0   11=252
gen  10       00=276   01=  0   10=  0   11=224
gen  20       00=347   01=  0   10=  0   11=153
gen  30       00=438   01=  0   10=  0   11= 62
gen  40       00=413   01=  0   10=  0   11= 87
gen  50       00=430   01=  0   10=  0   11= 70
gen  60       00=458   01=  0   10=  0   11= 42
gen  70       00=472   01=  0   10=  0   11= 28
gen  80       00=445   01=  0   10=  0   11= 55
gen  90       00=414   01=  0   10=  0   11= 86
gen 100       00=432   01=  0   10=  0   11= 68
gen 110       00=441   01=  0   10=  0   11= 59
gen 120       00=468   01=  0   10=  0   11= 32
gen 130       00=434   01=  0   10=  0   11= 66
gen 140       00=426   01=  0   10=  0   11= 74
gen 150       00=423   01=  0   10=  0   11= 77
gen 160       00=443   01=  0   10=  0   11= 57
gen 170       00=444   01=  0   10=  0   11= 56
gen 180       00=458   01=  0   10=  0   11= 42
gen 190       00=459   01=  0   10=  0   11= 41
gen 200       00=448   01=  0   10=  0   11= 52
gen 210       00=457   01=  0   10=  0   11= 43
gen 220       00=471   01=  0   10=  0   11= 29
gen 230       00=472   01=  0   10=  0   11= 28
gen 240       00=472   01=  0   10=  0   11= 28
gen 250       00=480   01=  0   10=  0   11= 20
gen 260       00=463   01=  0   10=  0   11= 37
gen 270       00=441   01=  0   10=  0   11= 59
gen 280       00=456   01=  0   10=  0   11= 44
gen 290       00=472   01=  0   10=  0   11= 28
gen 300       00=470   01=  0   10=  0   11= 30
gen 310       00=479   01=  0   10=  0   11= 21
gen 320       00=471   01=  0   10=  0   11= 29
gen 330       00=470   01=  0   10=  0   11= 30
gen 340       00=483   01=  0   10=  0   11= 17
gen 350       00=485   01=  0   10=  0   11= 15
```

gen 360	00=474	01=	0	10=	0	11=	26
gen 370	00=480	01=	0	10=	0	11=	20
gen 380	00=493	01=	0	10=	0	11=	7
gen 390	00=457	01=	0	10=	0	11=	43
gen 400	00=445	01=	0	10=	0	11=	55
gen 410	00=461	01=	0	10=	0	11=	39
gen 420	00=493	01=	0	10=	0	11=	7
gen 430	00=482	01=	0	10=	0	11=	18
gen 440	00=458	01=	0	10=	0	11=	42
gen 450	00=464	01=	0	10=	0	11=	36
gen 460	00=428	01=	0	10=	0	11=	72
gen 470	00=462	01=	0	10=	0	11=	38
gen 480	00=454	01=	0	10=	0	11=	46
gen 490	00=457	01=	0	10=	0	11=	43
gen 500	00=464	01=	0	10=	0	11=	36

A.4 Output for kin1.c: No Crossover

```
popsize=500 ipop=1 benefit=2 mute=1000 cross=0
gen   1       00=248   01=  0   10=  0   11=252
gen  10       00= 85   01=  3   10=  1   11=411
gen  20       00= 21   01=  5   10=  2   11=472
gen  30       00=  0   01=  1   10= 39   11=460
gen  40       00=  9   01=  3   10=202   11=286
gen  50       00=  2   01=  0   10=436   11= 62
gen  60       00=  6   01=  0   10=482   11= 12
gen  70       00= 27   01=  0   10=469   11=  4
gen  80       00= 22   01=  0   10=474   11=  4
gen  90       00= 16   01=  0   10=481   11=  3
gen 100       00=  9   01=  0   10=481   11= 10
gen 110       00= 15   01=  0   10=478   11=  7
gen 120       00=  0   01=  0   10=494   11=  6
gen 130       00=  1   01=  0   10=494   11=  5
gen 140       00=  2   01=  0   10=497   11=  1
gen 150       00=  2   01=  0   10=495   11=  3
gen 160       00=  0   01=  0   10=492   11=  8
gen 170       00=  0   01=  0   10=497   11=  3
gen 180       00=  5   01=  0   10=492   11=  3
gen 190       00= 17   01=  0   10=481   11=  2
gen 200       00= 33   01=  0   10=463   11=  4
gen 210       00= 12   01=  0   10=483   11=  5
gen 220       00= 14   01=  0   10=484   11=  2
gen 230       00=  5   01=  0   10=490   11=  5
gen 240       00=  1   01=  0   10=495   11=  4
gen 250       00= 11   01=  0   10=487   11=  2
gen 260       00= 41   01=  0   10=456   11=  3
gen 270       00= 45   01=  0   10=448   11=  7
gen 280       00= 25   01=  0   10=473   11=  2
gen 290       00= 45   01=  0   10=444   11= 11
gen 300       00= 23   01=  0   10=463   11= 14
gen 310       00= 37   01=  0   10=458   11=  5
gen 320       00= 84   01=  0   10=414   11=  2
gen 330       00= 79   01=  1   10=415   11=  5
gen 340       00= 73   01=  0   10=421   11=  6
gen 350       00= 64   01=  0   10=430   11=  6
gen 360       00= 64   01=  0   10=434   11=  2
gen 370       00= 66   01=  0   10=433   11=  1
```

gen 380	00=	38	01=	0	10=459	11=	3
gen 390	00=	16	01=	0	10=482	11=	2
gen 400	00=	22	01=	1	10=472	11=	5
gen 410	00=	12	01=	0	10=484	11=	4
gen 420	00=	21	01=	0	10=475	11=	4
gen 430	00=	18	01=	0	10=478	11=	4
gen 440	00=	37	01=	0	10=458	11=	5
gen 450	00=	40	01=	0	10=454	11=	6
gen 460	00=	60	01=	0	10=437	11=	3
gen 470	00=	56	01=	1	10=436	11=	7
gen 480	00=	62	01=	1	10=432	11=	5
gen 490	00=	26	01=	0	10=457	11=	17
gen 500	00=	28	01=	0	10=468	11=	4

Appendix B

'Green Beard' Program with Group Fitness: kin2

B.1 Source Code for kin2.c

```
/*
PROGRAM:   kin2.c
AUTHOR:    Tim Watson
MODIFIED: April 2003

Test to determine the properties of a GA-like population
with kin selective altruism ('Green Beard' Effect)
and group fitness.

   o Two-bit strings (beard colour, altruist)
   o Fixed probability that crossover will dislocate bits
   o Static population size (all generations equal size)
   o Proportion of 00, 01, 10, 11 in initial population fixed
   o Standard proportionate reproduction
   o One-point crossover
   o Statistics - number of 00s 01s, 10s and 11s in each gen
   o Both members of a pair receive the same fitness

Example: kin1 -p500 -g500 -c50 -i2 -b3 -m500 -r3
         popsize=500,
         generations=500,
         crossover prob=50%,
         initial population=type 2,
```

```
            benefit=3,
            muterate=1/500,
            random seed=3.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "r250.h"
#include "randlcg.h"


#define REPS        10000      /* used by pairs */
#define POPSIZE       500      /* must be even */
#define POPMAX       2000
#define CROSSOVER      50
#define INITPOP         1
#define BENEFIT         2
#define GENERATIONS  1000
#define MUTERATE     1000      /* 0 = no mutation */
#define RANDOMIZE       2


typedef struct member *Memptr;

typedef struct member {
  char    chrom[2];
  double  value;
  Memptr  lower,
          higher;
} Member;

Member curr[POPMAX], mate[POPMAX], next[POPMAX];


int counts[4];
```

```
/* precalced prob of n intact pairs after shuffling */
int pairs[12];
```

```
void init_pop(int, int, int);
void calc_fit(int, int);
void fitness(int, int);
void add_to_tree(Member *, Member *);
void update_stats(int);
void display_stats(int);
int prod_mate(Member *, int);
void copy_member(Member *, Member *);
void next_gen(int, int, int, int);



main(int argc, char *argv[])
{
int i, inc, popsize, gens, matesize, muterate, cross,\
    ipop, random, benefit;

popsize = POPSIZE;
matesize = POPSIZE;
gens = GENERATIONS;
cross = CROSSOVER;
ipop = INITPOP;
benefit = BENEFIT;
muterate = MUTERATE;
random = RANDOMIZE;
inc = 1;

while (--argc > 0 && (*++argv)[0] == '-')
switch (*++argv[0]) {
case 'p': if (*++argv[0])
            popsize = atoi(argv[0]);
          else if (--argc > 0)
            popsize = atoi(*++argv);
          break;
case 'g': if (*++argv[0])
```

```
            gens = atoi(argv[0]);
          else if (--argc > 0)
            gens = atoi(*++argv);
          break;
case 's': if (*++argv[0])
            inc = atoi(argv[0]);
          else if (--argc > 0)
            inc = atoi(*++argv);
          break;
case 'c': if (*++argv[0])
            cross = atoi(argv[0]);
          else if (--argc > 0)
            cross = atoi(*++argv);
          break;
case 'i': if (*++argv[0])
            ipop = atoi(argv[0]);
          else if (--argc > 0)
            ipop = atoi(*++argv);
          break;
case 'b': if (*++argv[0])
            benefit = atoi(argv[0]);
          else if (--argc > 0)
            benefit = atoi(*++argv);
          break;
case 'M': if (*++argv[0])
            matesize = atoi(argv[0]);
          else if (--argc > 0)
            matesize = atoi(*++argv);
          break;
case 'm': if (*++argv[0])
            muterate = atoi(argv[0]);
          else if (--argc > 0)
            muterate = atoi(*++argv);
          break;
case 'r': if (*++argv[0])
            random = atoi(argv[0]);
          else if (--argc > 0)
            random = atoi(*++argv);
          break;
default:  break;
}
```

```
if (popsize > 0) {
  if (popsize > POPMAX)
    popsize = POPMAX;
} else
  popsize = POPSIZE;

printf("popsize=%d ipop=%d benefit=%d mute=%d cross=%d\n",
       popsize, ipop, benefit, muterate, cross);

init_pop(popsize, random, ipop);

for (i=1; i<gens; i++) {
  calc_fit(popsize, benefit);
  if ((i/inc*inc==i) || (i==1))
    display_stats(i);
  prod_mate(curr, matesize);
  next_gen(matesize, popsize, muterate, cross);
}
```

```
  calc_fit(popsize, benefit);
  if (i/inc*inc==i)
    display_stats(i);

  return 0;
}
```

```
/* init_pop: create initial, random population */
void init_pop(int popsize, int random, int ipop)
{
  int i, j, halfpop, sum;
  double total;

  r250_init(random);

  /* precalculate how often i intact pairs will remain
     together after random shuffling, REPS times */
  halfpop = popsize>>1;
  pairs[0] = REPS>>1;
```

```
for (i=1; i<12; i++) {
  total = (double)REPS;
  for (j=0; j<i; j++)
    total *= (double)(halfpop-j) / \
                                   (double)((popsize-j)*(j+1));
  total *= ((double)halfpop/ \
            (double)(popsize-i)+(double)i) / (double)(i+1);
  pairs[i] = (int)(total+0.5);
}
sum = -pairs[0];
for (i=1; i<12; i++)
  sum += pairs[i];
pairs[0] -= sum;
for (i=1; i<12; i++)
  pairs[i] += pairs[i-1];

for (i=0; i<popsize; i++)
     switch (ipop) {
     case 1:  curr[i].chrom[0] = curr[i].chrom[1] = \
                                  '0' + r250() % 2;
              break;
     case 2:  curr[i].chrom[0] = '0' + r250() % 2;
              curr[i].chrom[1] = '0' + r250() % 2;
              break;
     default: break;
     }
}



void calc_fit(int popsize, int benefit)
{
  int i;

for (i=0; i<4; i++)
  counts[i] = 0;

fitness(0, benefit);
update_stats(0);
curr[0].lower = curr[0].higher = NULL;
curr[1].lower = curr[1].higher = NULL;
```

```
add_to_tree(curr, curr+1);

for (i=2; i<popsize-1; i+=2) {
  fitness(i, benefit);
  update_stats(i);
  curr[i].lower = curr[i].higher = NULL;
  curr[i+1].lower = curr[i+1].higher = NULL;
  add_to_tree(curr, curr+i);
  add_to_tree(curr, curr+i+1);
 }
}



/* fitness: return fitness for a pair */
void fitness(int i, int benefit)
{
  if ((curr[i].chrom[1]=='1')&&(curr[i+1].chrom[0]=='1')) {
    curr[i].value = curr[i+1].value = 3 + benefit;
  } else
    curr[i].value = curr[i+1].value = 4;
}



/* update_stats: update the counts */
void update_stats(int i)
{
  int count;

  count = (curr[i].chrom[0]-'0'<<1) + curr[i].chrom[1]-'0';
  counts[count]++;
  i++;
  count = (curr[i].chrom[0]-'0'<<1) + curr[i].chrom[1]-'0';
  counts[count]++;
}



/* add_to_tree: add <node> to <tree>, ordered by
   <node->value> */
```

```
void add_to_tree(Member *tree, Member *node)
{
  if (node->value > tree->value)
    if (tree->higher != NULL)
      add_to_tree(tree->higher, node);
    else
      tree->higher = node;
  else if (tree->lower != NULL)
      add_to_tree(tree->lower, node);
  else
      tree->lower = node;
}



/* display_stats: display the count of each of the four
   types of individual for the current generation */
void display_stats(int gen)
{
  printf("gen%4d       00=%3d   01=%3d   10=%3d   11=%3d\n",
          gen, counts[0], counts[1], counts[2], counts[3]);
}



/* prod_mate: recursively copy best <size> nodes from
   <curr> into <mate> (in reverse order) */
int prod_mate(Member *tree, int size)
{
  if (tree != NULL) {
    if (tree->higher != NULL)
      size = prod_mate(tree->higher, size);
    if (size > 0) {
      copy_member(mate+size-1, tree);
      size = prod_mate(tree->lower, size -= 1);
    }
  }

  return size;
}
```

```
/* copy_member: copy member from <curr> to <mate>
   (destroys tree) */
void copy_member(Member *to, Member *from)
{
  strcpy(to->chrom, from->chrom);
  to->value = from->value;
}



/* next_gen: reproduce, crossover and mutate
   from <mate> to <curr> via <next> */
void next_gen(int size, int popsize, int mute, int cross)
{
  int i, j, n;
  static double choice;
  static char temp;

for (i=1; i<size; i++)     /* calculate proportions */
  mate[i].value += mate[i-1].value;

for (i=0; i<popsize; i++) {              /* reproduce */
  choice = r250() % (int) mate[size-1].value + 1;
  j = 0;
  while (j < size-1 && choice > mate[j].value)
    j++;
  strcpy((next+i)->chrom, (mate+j)->chrom);
}

for (i=0; i<popsize; i+=2)           /* crossover */
  if (r250()%100 < cross) {
    temp = next[i].chrom[0];
    next[i].chrom[0] = next[i+1].chrom[0];
    next[i+1].chrom[0] = temp;
  }

if (mute != 0)
  for (i=0; i<popsize; i++)                    /* mutation */
    for (j=0; j<2; j++)
```

```
      if ((r250() % mute) == 0)
        next[i].chrom[j] = '0' + '1' - next[i].chrom[j];

 /* simulate random shuffling from next back to
    current generation */
n = r250() % REPS + 1;
  for (i=0; n>pairs[i>>1]; i+=2) {
    strcpy((curr+i)->chrom, (next+i)->chrom);
    strcpy((curr+i+1)->chrom, (next+i+1)->chrom);
  }
  strcpy((curr+popsize-1)->chrom, (next+i)->chrom);
  for (i++; i<popsize-1; i+=2) {
    strcpy((curr+i-1)->chrom, (next+i)->chrom);
    strcpy((curr+i)->chrom, (next+i+1)->chrom);
  }
}
```

B.2 Output for kin2.c: No Crossover

```
popsize=500 ipop=1 benefit=2 mute=1000 cross=0
gen   1       00=248   01=  0   10=  0   11=252
gen  10       00= 74   01=  3   10=  0   11=423
gen  20       00= 12   01=  5   10=  3   11=480
gen  30       00=  3   01=  8   10=  2   11=487
gen  40       00=  0   01=  8   10=  0   11=492
gen  50       00=  0   01=  4   10=  3   11=493
gen  60       00=  0   01=  6   10=  1   11=493
gen  70       00=  0   01=  3   10=  1   11=496
gen  80       00=  0   01=  4   10=  0   11=496
gen  90       00=  0   01=  0   10=  0   11=500
gen 100       00=  0   01=  0   10=  4   11=496
gen 110       00=  0   01= 17   10=  2   11=481
gen 120       00=  0   01=  0   10=  3   11=497
gen 130       00=  0   01=  6   10=  3   11=491
gen 140       00=  0   01=  2   10=  6   11=492
gen 150       00=  0   01=  4   10=  0   11=496
gen 160       00=  0   01=  0   10=  2   11=498
gen 170       00=  0   01= 11   10=  8   11=481
gen 180       00=  0   01=  1   10=  1   11=498
gen 190       00=  0   01=  0   10=  6   11=494
gen 200       00=  0   01=  1   10=  1   11=498
gen 210       00=  0   01=  2   10=  1   11=497
gen 220       00=  0   01= 14   10=  1   11=485
gen 230       00=  0   01= 12   10=  0   11=488
gen 240       00=  0   01= 19   10=  3   11=478
gen 250       00=  0   01=  0   10=  0   11=500
gen 260       00=  0   01=  5   10=  2   11=493
gen 270       00=  0   01=  2   10=  6   11=492
gen 280       00=  0   01=  3   10=  1   11=496
gen 290       00=  0   01=  0   10=  2   11=498
gen 300       00=  0   01=  5   10=  5   11=490
gen 310       00=  0   01=  6   10= 14   11=480
gen 320       00=  0   01= 13   10=  8   11=479
gen 330       00=  0   01= 15   10= 17   11=468
gen 340       00=  0   01=  3   10= 11   11=486
gen 350       00=  0   01=  3   10=  2   11=495
gen 360       00=  0   01=  1   10=  1   11=498
gen 370       00=  0   01=  2   10=  0   11=498
```

gen 380	00=	0	01=	1	10=	0	11=499
gen 390	00=	0	01=	0	10=	0	11=500
gen 400	00=	0	01=	0	10=	6	11=494
gen 410	00=	0	01=	3	10=	2	11=495
gen 420	00=	0	01=	9	10=	2	11=489
gen 430	00=	0	01=	6	10=	1	11=493
gen 440	00=	0	01=	2	10=	0	11=498
gen 450	00=	0	01=	5	10=	0	11=495
gen 460	00=	0	01=	0	10=	1	11=499
gen 470	00=	0	01=	0	10=	3	11=497
gen 480	00=	0	01=	3	10=	3	11=494
gen 490	00=	0	01=	0	10=	6	11=494
gen 500	00=	0	01=	1	10=	0	11=499

B.3 Output for kin2.c: No Crossover and no Benefit

```
popsize=500 ipop=1 benefit=0 mute=1000 cross=0
gen   1       00=248   01=  0   10=  0   11=252
gen  10       00=344   01=  1   10=  7   11=148
gen  20       00=349   01=  5   10= 54   11= 92
gen  30       00=352   01=  0   10= 59   11= 89
gen  40       00=392   01=  6   10= 59   11= 43
gen  50       00=388   01=  2   10= 71   11= 39
gen  60       00=405   01= 12   10= 37   11= 46
gen  70       00=444   01=  9   10= 11   11= 36
gen  80       00=428   01=  0   10= 18   11= 54
gen  90       00=421   01=  0   10= 21   11= 58
gen 100       00=453   01=  0   10=  7   11= 40
gen 110       00=413   01=  7   10= 53   11= 27
gen 120       00=399   01=  8   10= 62   11= 31
gen 130       00=381   01= 15   10= 79   11= 25
gen 140       00=406   01=  4   10= 73   11= 17
gen 150       00=414   01=  5   10= 62   11= 19
gen 160       00=443   01= 16   10= 22   11= 19
gen 170       00=422   01= 31   10= 26   11= 21
gen 180       00=433   01= 10   10= 10   11= 47
gen 190       00=430   01= 10   10=  5   11= 55
gen 200       00=432   01= 21   10=  2   11= 45
gen 210       00=467   01=  7   10=  1   11= 25
gen 220       00=441   01=  7   10=  7   11= 45
gen 230       00=442   01= 19   10= 11   11= 28
gen 240       00=440   01=  5   10=  7   11= 48
gen 250       00=460   01=  2   10= 10   11= 28
gen 260       00=452   01=  3   10=  9   11= 36
gen 270       00=431   01=  0   10= 13   11= 56
gen 280       00=400   01=  7   10= 11   11= 82
gen 290       00=427   01= 11   10= 17   11= 45
gen 300       00=423   01= 11   10= 17   11= 49
gen 310       00=401   01= 21   10= 43   11= 35
gen 320       00=409   01= 16   10= 38   11= 37
gen 330       00=407   01= 46   10= 26   11= 21
gen 340       00=411   01= 53   10= 27   11=  9
gen 350       00=420   01= 26   10= 44   11= 10
```

gen 360	00=420	01= 34	10= 17	11= 29
gen 370	00=393	01= 58	10= 25	11= 24
gen 380	00=387	01= 23	10= 70	11= 20
gen 390	00=430	01= 9	10= 41	11= 20
gen 400	00=400	01= 26	10= 64	11= 10
gen 410	00=385	01= 39	10= 63	11= 13
gen 420	00=373	01= 50	10= 57	11= 20
gen 430	00=397	01= 59	10= 30	11= 14
gen 440	00=420	01= 26	10= 45	11= 9
gen 450	00=400	01= 46	10= 37	11= 17
gen 460	00=409	01= 68	10= 9	11= 14
gen 470	00=376	01= 68	10= 40	11= 16
gen 480	00=298	01= 86	10= 85	11= 31
gen 490	00=288	01= 34	10=155	11= 23
gen 500	00=299	01= 36	10=133	11= 32

B.4 Output for kin2.c: Both Crossover and Mutation

```
popsize=500 ipop=1 benefit=2 mute=1000 cross=50
gen   1       00=248   01=  0   10=  0   11=252
gen  10       00= 53   01=118   10= 94   11=235
gen  20       00=  8   01= 45   10= 56   11=391
gen  30       00=  3   01= 54   10= 22   11=421
gen  40       00=  1   01= 16   10= 21   11=462
gen  50       00=  0   01= 15   10= 14   11=471
gen  60       00=  0   01= 26   10=  2   11=472
gen  70       00=  0   01= 15   10=  8   11=477
gen  80       00=  0   01= 14   10=  0   11=486
gen  90       00=  0   01=  0   10=  1   11=499
gen 100       00=  0   01=  0   10=  4   11=496
gen 110       00=  0   01=  5   10=  4   11=491
gen 120       00=  0   01=  1   10=  2   11=497
gen 130       00=  0   01=  2   10=  1   11=497
gen 140       00=  0   01=  2   10=  4   11=494
gen 150       00=  0   01=  2   10=  0   11=498
gen 160       00=  0   01=  0   10=  2   11=498
gen 170       00=  0   01=  3   10=  7   11=490
gen 180       00=  0   01=  1   10=  2   11=497
gen 190       00=  0   01=  0   10=  6   11=494
gen 200       00=  0   01=  1   10=  1   11=498
gen 210       00=  0   01=  3   10=  9   11=488
gen 220       00=  0   01=  3   10= 10   11=487
gen 230       00=  0   01=  3   10= 10   11=487
gen 240       00=  0   01=  0   10=  7   11=493
gen 250       00=  0   01=  1   10= 14   11=485
gen 260       00=  2   01=  8   10= 15   11=475
gen 270       00=  0   01=  2   10=  1   11=497
gen 280       00=  0   01=  0   10=  1   11=499
gen 290       00=  0   01=  6   10=  4   11=490
gen 300       00=  0   01=  1   10=  2   11=497
gen 310       00=  1   01=  7   10= 15   11=477
gen 320       00=  0   01= 16   10= 13   11=471
gen 330       00=  0   01= 14   10=  6   11=480
gen 340       00=  0   01= 13   10= 11   11=476
gen 350       00=  1   01= 19   10=  4   11=476
```

gen 360	00=	0	01=	6	10=	0	11=494
gen 370	00=	0	01=	3	10=	0	11=497
gen 380	00=	0	01=	4	10=	0	11=496
gen 390	00=	0	01=	3	10=	0	11=497
gen 400	00=	0	01=	0	10=	6	11=494
gen 410	00=	1	01=	30	10=	8	11=461
gen 420	00=	0	01=	15	10=	13	11=472
gen 430	00=	0	01=	12	10=	9	11=479
gen 440	00=	0	01=	7	10=	11	11=482
gen 450	00=	0	01=	1	10=	7	11=492
gen 460	00=	0	01=	0	10=	2	11=498
gen 470	00=	0	01=	5	10=	15	11=480
gen 480	00=	0	01=	3	10=	10	11=487
gen 490	00=	0	01=	4	10=	11	11=485
gen 500	00=	0	01=	8	10=	6	11=486

Appendix C

'Green Beard' Program with Group Fitness Bit: kin3

C.1 Source Code for kin3.c

```
/*
PROGRAM:   kin3.c
AUTHOR:    Tim Watson
MODIFIED: April 2003

Test to determine the properties of a GA-like population
with kin selective altruism ('Green Beard' Effect)
and with an extra 'group fitness' bit.

  o Three-bit strings (beard colour, altruist, groupfit)
  o Fixed probability that crossover will dislocate bits
  o Fixed prob that if crossover occurs it's between bit 2&3
  o Static population size (all generations equal size)
  o Proportion of 00x, 01x, 10x, 11x in init pop fixed
  o Standard proportionate reproduction
  o One-point crossover
  o Statistics - number of 00xs, 01xs, 10xs and 11xs
      in each gen (x=0s on left x=1s on right when displayed)

Example: kin1 -p500 -g500 -c50 -i2 -b3 -m500 -r3
         popsize=500,
         generations=500,
         crossover prob=50%,
```

```
            initial population=type 2,
            benefit=3,
            muterate=1/500,
            random seed=3.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "r250.h"
#include "randlcg.h"


#define REPS        10000      /* used by pairs */
#define POPSIZE       500      /* must be even */
#define POPMAX       2000
#define CROSSOVER      50
#define BIT3CROSS      50
#define INITPOP         1
#define BENEFIT         2
#define GENERATIONS  1000
#define MUTERATE     1000      /* 0 = no mutation */
#define RANDOMIZE       2


typedef struct member *Memptr;

typedef struct member {
  char    chrom[3];
  double  value;
  Memptr  lower,
          higher;
} Member;

Member curr[POPMAX], mate[POPMAX], next[POPMAX];
```

```
int counts[8];

/* precalced prob of n intact pairs after shuffling */
int pairs[12];
```

```
void init_pop(int, int, int);
void calc_fit(int, int);
void fitness(int, int);
void add_to_tree(Member *, Member *);
void update_stats(int);
void display_stats(int);
int prod_mate(Member *, int);
void copy_member(Member *, Member *);
void next_gen(int, int, int, int, int);
```

```
main(int argc, char *argv[])
{
  int i, inc, popsize, gens, matesize, muterate, cross,\
      b3cross, ipop, random, benefit;

  popsize = POPSIZE;
  matesize = POPSIZE;
  gens = GENERATIONS;
  cross = CROSSOVER;
  b3cross = BIT3CROSS;
  ipop = INITPOP;
  benefit = BENEFIT;
  muterate = MUTERATE;
  random = RANDOMIZE;
  inc = 1;

while (--argc > 0 && (*++argv)[0] == '-')
  switch (*++argv[0]) {
  case 'p': if (*++argv[0])
              popsize = atoi(argv[0]);
            else if (--argc > 0)
```

```
            popsize = atoi(*++argv);
          break;
case 'g': if (*++argv[0])
            gens = atoi(argv[0]);
          else if (--argc > 0)
            gens = atoi(*++argv);
          break;
case 's': if (*++argv[0])
            inc = atoi(argv[0]);
          else if (--argc > 0)
            inc = atoi(*++argv);
          break;
case 'c': if (*++argv[0])
            cross = atoi(argv[0]);
          else if (--argc > 0)
            cross = atoi(*++argv);
          break;
case 'C': if (*++argv[0])
            b3cross = atoi(argv[0]);
          else if (--argc > 0)
            b3cross = atoi(*++argv);
          break;
case 'i': if (*++argv[0])
            ipop = atoi(argv[0]);
          else if (--argc > 0)
            ipop = atoi(*++argv);
          break;
case 'b': if (*++argv[0])
            benefit = atoi(argv[0]);
          else if (--argc > 0)
            benefit = atoi(*++argv);
          break;
case 'M': if (*++argv[0])
            matesize = atoi(argv[0]);
          else if (--argc > 0)
            matesize = atoi(*++argv);
          break;
case 'm': if (*++argv[0])
            muterate = atoi(argv[0]);
          else if (--argc > 0)
            muterate = atoi(*++argv);
```

```
            break;
case 'r': if (*++argv[0])
            random = atoi(argv[0]);
          else if (--argc > 0)
            random = atoi(*++argv);
          break;
  default:  break;
  }

if (popsize > 0) {
  if (popsize > POPMAX)
    popsize = POPMAX;
} else
  popsize = POPSIZE;

printf("pop=%d ipop=%d ben=%d mute=%d cross=%d b3x=%d\n",
       popsize, ipop, benefit, muterate, cross, b3cross);

init_pop(popsize, random, ipop);

  for (i=1; i<gens; i++) {
    calc_fit(popsize, benefit);
    if ((i/inc*inc==i) || (i==1))
      display_stats(i);
    prod_mate(curr, matesize);
    next_gen(matesize, popsize, muterate, cross, b3cross);
  }

  calc_fit(popsize, benefit);
  if (i/inc*inc==i)
    display_stats(i);

  return 0;
}
```

```
/* init_pop: create initial, random population */
void init_pop(int popsize, int random, int ipop)
{
  int i, j, halfpop, sum;
```

```
double total;

r250_init(random);

/* precalculate how often i intact pairs will remain
   together after random shuffling, REPS times */
halfpop = popsize>>1;
pairs[0] = REPS>>1;
for (i=1; i<12; i++) {
  total = (double)REPS;
  for (j=0; j<i; j++)
    total *= (double)(halfpop-j) / \
                                (double)((popsize-j)*(j+1));
  total *= ((double)halfpop/ \
            (double)(popsize-i)+(double)i) / (double)(i+1);
  pairs[i] = (int)(total+0.5);
}
sum = -pairs[0];
for (i=1; i<12; i++)
  sum += pairs[i];
pairs[0] -= sum;
for (i=1; i<12; i++)
  pairs[i] += pairs[i-1];

for (i=0; i<popsize; i++) {
  curr[i].chrom[2] = '0';
     switch (ipop) {
     case 1:  curr[i].chrom[0] = curr[i].chrom[1] = \
                                    '0' + r250() % 2;
              break;
  case 2:  curr[i].chrom[0] = '0' + r250() % 2;
           curr[i].chrom[1] = '0' + r250() % 2;
           break;
  default: break;
     }
  }
}


void calc_fit(int popsize, int benefit)
```

```
{
  int i;

  for (i=0; i<8; i++)
    counts[i] = 0;
fitness(0, benefit);
update_stats(0);
curr[0].lower = curr[0].higher = NULL;
curr[1].lower = curr[1].higher = NULL;
add_to_tree(curr, curr+1);

for (i=2; i<popsize-1; i+=2) {
  fitness(i, benefit);
  update_stats(i);
  curr[i].lower = curr[i].higher = NULL;
  curr[i+1].lower = curr[i+1].higher = NULL;
  add_to_tree(curr, curr+i);
  add_to_tree(curr, curr+i+1);
}
}



/* fitness: return fitness for a pair */
void fitness(int i, int benefit)
{
  curr[i].value = 4;
  curr[i+1].value = 4;
  /* The next if statement is equivalent to: if the donor
     uses groupfit then it will only donate if the
     recipient accepts groupfit */
  if (curr[i].chrom[2] == '0') {
    if ((curr[i].chrom[1]=='1') \
                          && (curr[i+1].chrom[0]=='1')) {
    curr[i].value = 2;
    curr[i+1].value = 4 + (benefit<<1);
  }
} else if ((curr[i].chrom[1]=='1') \
                  && (curr[i+1].chrom[0]=='1') \
                          && (curr[i+1].chrom[2]=='1'))
```

```
    curr[i].value = curr[i+1].value = 3 + benefit;
}


/* update_stats: update the counts */
void update_stats(int i)
{
  int count;

  count = (curr[i].chrom[0]-'0'<<2) + \
          (curr[i].chrom[1]-'0'<<1) + curr[i].chrom[2]-'0';
  counts[count]++;
  i++;
  count = (curr[i].chrom[0]-'0'<<2) + \
          (curr[i].chrom[1]-'0'<<1) + curr[i].chrom[2]-'0';
  counts[count]++;
}


/* add_to_tree: add <node> to <tree>, ordered by
   <node->value> */
void add_to_tree(Member *tree, Member *node)
{
  if (node->value > tree->value)
    if (tree->higher != NULL)
      add_to_tree(tree->higher, node);
    else
      tree->higher = node;
  else if (tree->lower != NULL)
      add_to_tree(tree->lower, node);
  else
      tree->lower = node;
}


/* display_stats: display the count of each of the eight
   types of individual for the current generation */
void display_stats(int gen)
```

```
{
  printf("%4d %3d %3d %3d %3d groupfit-> %3d %3d %3d %3d\n",
         gen, counts[0], counts[2], counts[4], counts[6],
         counts[1], counts[3], counts[5], counts[7]);
}
```

```
/* prod_mate: recursively copy best <size> nodes from
   <curr> into <mate> (in reverse order) */
int prod_mate(Member *tree, int size)
{
  if (tree != NULL) {
    if (tree->higher != NULL)
      size = prod_mate(tree->higher, size);
    if (size > 0) {
      copy_member(mate+size-1, tree);
      size = prod_mate(tree->lower, size -= 1);
    }
  }
  return size;
}
```

```
/* copy_member: copy member from <curr> to <mate>
   (destroys tree) */
void copy_member(Member *to, Member *from)
{
  strcpy(to->chrom, from->chrom);
  to->value = from->value;
}
```

```
/* next_gen: reproduce, crossover and mutate
   from <mate> to <curr> via <next> */
void next_gen(int size, int pop, int mute, int cross, int b)
{
  int i, j, n;
```

```
static double choice;
static char temp;

for (i=1; i<size; i++)      /* calculate proportions */
  mate[i].value += mate[i-1].value;

for (i=0; i<pop; i++) {                  /* reproduce */
  choice = r250() % (int) mate[size-1].value + 1;
  j = 0;
  while (j < size-1 && choice > mate[j].value)
    j++;
  strcpy((next+i)->chrom, (mate+j)->chrom);
}

for (i=0; i<pop; i+=2)              /* crossover */
  if (r250()%100 < cross)
    if (r250()%100 < b) {
      temp = next[i].chrom[2];
      next[i].chrom[2] = next[i+1].chrom[2];
      next[i+1].chrom[2] = temp;
    } else {
      temp = next[i].chrom[0];
      next[i].chrom[0] = next[i+1].chrom[0];
      next[i+1].chrom[0] = temp;
    }

if (mute != 0)
  for (i=0; i<pop; i++)                    /* mutation */
    for (j=0; j<3; j++)
      if ((r250() % mute) == 0)
        next[i].chrom[j] = '0' + '1' - next[i].chrom[j];

 /* simulate random shuffling from next back to
    current generation */
n = r250() % REPS + 1;
for (i=0; n>pairs[i>>1]; i+=2) {
  strcpy((curr+i)->chrom, (next+i)->chrom);
  strcpy((curr+i+1)->chrom, (next+i+1)->chrom);
}
strcpy((curr+pop-1)->chrom, (next+i)->chrom);
for (i++; i<pop-1; i+=2) {
```

```
      strcpy((curr+i-1)->chrom, (next+i)->chrom);
      strcpy((curr+i)->chrom, (next+i+1)->chrom);
    }
  }
```

C.2 Output for kin3.c: 1000 Generations

```
pop=500 ipop=1 ben=2 mute=1000 cross=50 b3x=50
   1 248   0   0 252 groupfit->   0   0   0   0
  10 114  29 259  79 groupfit->   6   4   8   1
  20  42   4 379  38 groupfit->   2   1  33   1
  30 104   4 338  13 groupfit->  10   0  28   3
  40  58   5 362  15 groupfit->   5   0  41  14
  50  56   0 365   2 groupfit->  12   0  64   1
  60  58   0 324   5 groupfit->  24   0  89   0
  70  34   0 333   5 groupfit->  11   0 117   0
  80  42   0 394   2 groupfit->   7   0  55   0
  90  45   0 387   7 groupfit->  13   0  47   1
 100  69   1 352   4 groupfit->  12   0  62   0
 110  35   1 342   1 groupfit->  18   0 102   1
 120  59   0 341  11 groupfit->  12   0  73   4
 130  50   1 379   4 groupfit->   8   0  58   0
 140  33   0 446   2 groupfit->   3   0  16   0
 150  31   0 436   5 groupfit->   2   0  26   0
 160  13   0 474   2 groupfit->   3   0   8   0
 170  33   0 453   4 groupfit->   0   0  10   0
 180  58   0 428   7 groupfit->   2   0   5   0
 190  40   0 452   3 groupfit->   0   0   5   0
 200  34   0 456   6 groupfit->   1   0   2   1
 210  34   0 463   3 groupfit->   0   0   0   0
 220  12   0 480   3 groupfit->   0   0   5   0
 230   3   0 437   5 groupfit->   0   0  55   0
 240  28   1 399   6 groupfit->   9   0  57   0
 250  32   0 430   1 groupfit->   3   0  33   1
 260  24   0 453   3 groupfit->   1   0  19   0
 270  32   0 455   4 groupfit->   0   0   9   0
 280   9   0 476   6 groupfit->   0   0   9   0
 290  11   0 457  13 groupfit->   0   0  19   0
 300  16   0 453   8 groupfit->   0   0  23   0
 310  20   0 471   4 groupfit->   0   0   5   0
 320  10   0 455   1 groupfit->   2   0  32   0
 330   8   0 471   4 groupfit->   0   0  17   0
 340   6   0 479   5 groupfit->   0   0  10   0
 350   0   0 468  12 groupfit->   0   0  19   1
 360   8   0 471   6 groupfit->   0   0  15   0
 370   6   0 466   9 groupfit->   0   0  19   0
```

380	5	0	460	10	groupfit->	0	0	24	1
390	1	0	432	2	groupfit->	0	0	65	0
400	4	0	422	6	groupfit->	1	0	67	0
410	7	1	433	4	groupfit->	1	0	54	0
420	2	0	413	1	groupfit->	1	0	83	0
430	0	0	412	2	groupfit->	0	0	85	1
440	4	0	414	5	groupfit->	0	0	75	2
450	10	0	416	1	groupfit->	2	0	71	0
460	17	0	364	6	groupfit->	2	1	108	2
470	35	0	382	7	groupfit->	7	0	61	8
480	27	3	396	2	groupfit->	1	0	66	5
490	22	1	392	1	groupfit->	3	0	81	0
500	17	0	399	4	groupfit->	4	0	75	1
510	4	0	404	5	groupfit->	0	0	87	0
520	0	0	386	2	groupfit->	1	0	110	1
530	1	0	305	4	groupfit->	1	0	187	2
540	0	0	390	1	groupfit->	0	0	109	0
550	5	0	362	5	groupfit->	1	0	125	2
560	6	0	335	4	groupfit->	6	0	148	1
570	10	0	269	8	groupfit->	4	0	204	5
580	9	0	270	6	groupfit->	12	0	195	8
590	17	0	311	16	groupfit->	5	0	144	7
600	8	2	304	7	groupfit->	1	0	175	3
610	21	0	286	4	groupfit->	9	0	179	1
620	14	0	246	5	groupfit->	19	0	205	11
630	10	3	225	4	groupfit->	8	0	245	5
640	4	0	251	8	groupfit->	7	0	229	1
650	0	0	194	1	groupfit->	0	0	303	2
660	0	0	168	1	groupfit->	4	0	322	5
670	2	0	248	4	groupfit->	7	0	231	8
680	0	0	238	10	groupfit->	0	0	244	8
690	1	0	189	8	groupfit->	2	0	281	19
700	0	0	221	6	groupfit->	0	0	264	9
710	5	0	291	8	groupfit->	3	0	193	0
720	1	0	346	5	groupfit->	2	0	142	4
730	4	0	339	3	groupfit->	3	0	150	1
740	7	0	356	1	groupfit->	1	0	132	3
750	1	0	335	19	groupfit->	0	0	138	7
760	6	0	299	11	groupfit->	0	0	178	6
770	9	0	271	11	groupfit->	1	0	189	19
780	3	0	251	6	groupfit->	3	0	235	2

```
790    1    0 224    8 groupfit->    1    0 247   19
800    2    0 167    6 groupfit->    9    0 298   18
810    2    1 134    4 groupfit->    4    3 303   49
820    0    0  94    5 groupfit->    9    5 325   62
830    2    0  35    5 groupfit->   15    2 322  119
840    0    0   9   10 groupfit->    4    2 242  233
850    0    0   5    3 groupfit->    0    1 150  341
860    0    0   5   10 groupfit->    0    1  73  411
870    0    0   0    1 groupfit->    0    9  17  473
880    0    0   0    6 groupfit->    0   11   3  480
890    0    0   0    5 groupfit->    0    5   3  487
900    0    0   0    3 groupfit->    0    5   0  492
910    0    0   0    4 groupfit->    0    3  10  483
920    0    0   0    1 groupfit->    0    1   1  497
930    0    0   0    4 groupfit->    0    7   2  487
940    0    0   0    3 groupfit->    0    2   6  489
950    0    0   0    6 groupfit->    0    2  11  481
960    0    0   0    1 groupfit->    0    0   2  497
970    0    0   0    3 groupfit->    0    0   4  493
980    0    0   0    3 groupfit->    0   16   1  480
990    0    0   0    4 groupfit->    0    6   4  486
1000   0    0   0    3 groupfit->    0    9   0  488
```

C.3 Output for kin3.c: 9999 Generations

```
pop=500 ipop=1 ben=2 mute=1000 cross=50 b3x=50
   1 248   0   0 252 groupfit->   0   0   0   0
 100  69   1 352   4 groupfit->  12   0  62   0
 200  34   0 456   6 groupfit->   1   0   2   1
 300  16   0 453   8 groupfit->   0   0  23   0
 400   4   0 422   6 groupfit->   1   0  67   0
 500  17   0 399   4 groupfit->   4   0  75   1
 600   8   2 304   7 groupfit->   1   0 175   3
 700   0   0 221   6 groupfit->   0   0 264   9
 800   2   0 167   6 groupfit->   9   0 298  18
 900   0   0   0   3 groupfit->   0   5   0 492
1000   0   0   0   3 groupfit->   0   9   0 488
1100   0   0   0   5 groupfit->   0   0   0 495
1200   0   0   0   3 groupfit->   0   3   7 487
1300   0   0   0   2 groupfit->   0   5  14 479
1400   0   0   0   2 groupfit->   0  12   6 480
1500   0   0   0   2 groupfit->   0   8   1 489
1600   0   0   1   3 groupfit->   0   1   7 488
1700   0   0   0   4 groupfit->   0   2   3 491
1800   0   0   0   2 groupfit->   0   7   1 490
1900   0   0   0   2 groupfit->   0   3   4 491
2000   0   0   0   6 groupfit->   0  15   4 475
2100   0   0   1  12 groupfit->   0   2  11 474
2200   0   0   0   2 groupfit->   0   0   2 496
2300   0   0   0   3 groupfit->   0   3   3 491
2400   0   0   0   3 groupfit->   0   5   2 490
2500   0   0   0   3 groupfit->   0   2   8 487
2600   0   0   0   5 groupfit->   0   1   2 492
2700   0   0   0   3 groupfit->   0   7   6 484
2800   0   0   0   5 groupfit->   0   1   1 493
2900   0   0   0   3 groupfit->   0   1   0 496
3000   0   0   0   2 groupfit->   0   1   7 490
3100   0   0   0   4 groupfit->   0   0   4 492
3200   0   0   0   4 groupfit->   0  13   4 479
3300   0   0   0   3 groupfit->   0   2   0 495
3400   0   0   0   3 groupfit->   0   2   2 493
3500   0   0   0   4 groupfit->   0   2   4 490
3600   0   0   0   3 groupfit->   1   4  18 474
3700   0   0   0   4 groupfit->   0   5   1 490
```

```
3800    0    0    0    3 groupfit->    0   11    2 484
3900    0    0    0    9 groupfit->    0    7    2 482
4000    0    0    0    2 groupfit->    0    0   10 488
4100    0    0    0    6 groupfit->    0    1    9 484
4200    0    0    0    1 groupfit->    0    7    5 487
4300    0    0    0    2 groupfit->    0   14    2 482
4400    0    0    0    2 groupfit->    0    5   10 483
4500    0    0    0    1 groupfit->    0    3    8 488
4600    0    0    0    8 groupfit->    0    8    1 483
4700    0    0    0    2 groupfit->    0    0    1 497
4800    0    0    0    4 groupfit->    0    8    1 487
4900    0    0    0    3 groupfit->    0    2    2 493
5000    0    0    0    3 groupfit->    0    2    1 494
5100    0    0    0    5 groupfit->    0    0    2 493
5200    0    0    0    4 groupfit->    0    1    1 494
5300    0    0    0    3 groupfit->    0    4    2 491
5400    0    0    0    5 groupfit->    0   14    4 477
5500    0    0    0    3 groupfit->    0    4    0 493
5600    0    0    0    2 groupfit->    0    3    9 486
5700    0    0    0    8 groupfit->    0    0    0 492
5800    0    0    0    5 groupfit->    0    2    0 493
5900    0    0    0    3 groupfit->    0    2   27 468
6000    0    0    0    2 groupfit->    0    0    5 493
6100    0    0    0    3 groupfit->    0    0    2 495
6200    0    0    0    3 groupfit->    0    5   30 462
6300    0    0    3    8 groupfit->    0    9    8 472
6400    0    0    0    2 groupfit->    0    0    1 497
6500    0    0    0    1 groupfit->    0    7    1 491
6600    0    0    0    7 groupfit->    1    3    8 481
6700    0    0    0    4 groupfit->    0    0    0 496
6800    0    0    0    2 groupfit->    0    2    7 489
6900    0    0    0    2 groupfit->    0    1   18 479
7000    0    0    0    2 groupfit->    0    2    8 488
7100    0    0    0    2 groupfit->    0    9    5 484
7200    0    0    0    2 groupfit->    0    3   10 485
7300    0    0    0    7 groupfit->    0    1    1 491
7400    0    0    0    2 groupfit->    0    4    4 490
7500    0    0    0    1 groupfit->    0    1    3 495
7600    0    0    0    3 groupfit->    0    7    2 488
7700    0    0    0    1 groupfit->    0    3   14 482
7800    0    0    0    3 groupfit->    0    4    1 492
```

7900	0	0	0	4	groupfit->	0	2	1	493
8000	0	0	0	2	groupfit->	0	7	12	479
8100	0	0	0	2	groupfit->	0	2	3	493
8200	0	0	0	2	groupfit->	0	0	3	495
8300	0	0	1	4	groupfit->	1	9	51	434
8400	0	0	0	3	groupfit->	0	2	0	495
8500	0	0	0	1	groupfit->	0	6	0	493
8600	0	0	0	2	groupfit->	0	9	8	481
8700	0	0	0	6	groupfit->	0	2	6	486
8800	0	0	0	4	groupfit->	0	14	6	476
8900	0	0	0	2	groupfit->	0	2	13	483
9000	0	0	1	1	groupfit->	0	1	6	491
9100	0	0	0	8	groupfit->	0	1	5	486
9200	0	0	0	2	groupfit->	0	9	0	489
9300	0	0	0	6	groupfit->	0	1	1	492
9400	0	0	0	3	groupfit->	0	3	3	491
9500	0	0	0	3	groupfit->	0	4	3	490
9600	0	0	0	3	groupfit->	0	0	1	496
9700	0	0	0	2	groupfit->	0	4	11	483
9800	0	1	0	4	groupfit->	0	8	6	481
9900	0	0	0	5	groupfit->	0	3	3	489

Appendix D

Shuffling Simulation Test Program: pairs

For reasons of computational efficiency, the random partitioning of a population into pairs was simulated in a number of the experimental programs (see the source code in Appendices A–C) by an algorithm that uses the results of the calculations performed by the code in the test program in Section D.1.

The random pairing simulation algorithm works by randomly choosing n, with the likelihoods of the different values for n determined by the precalculated values, such that n pairs will stay together in the population. The algorithm then copies n pairs, intact, from the previous to the next generation. Since the programs that use it divide the population into pairs by putting the first two individuals in the population into the first pair, the next two into the second and so on, the code splits up the remaining pairs by copying the first individual in the first pair to be split to the bottom of the next generation array, and then shifting the remaining individuals up one as they are copied across from the previous to the next generation.

The output in Section D.2 shows that a minor adjustment has to be

made for some population sizes so that the sum of the likelihoods always equals 10,000 (the numbers represent how many populations would contain n intact pairs if 10,000 populations were randomly paired). As can be seen, this correction is either 0 or -1 and is performed by adjusting the largest likelihood, which is always that no pairs will remain intact. Overall, the simulation is indistinguishable in effect from an algorithm that uses a PRNG to perform the repartitioning but is considerably faster.

D.1 Source Code for pairs.c

```
/*
PROGRAM:  pairs.c
AUTHOR:   Tim Watson
MODIFIED: April 2003

A program to calculate the probablilty of pairs
staying together after random shuffling.
*/

#include <stdio.h>

#define REPS   10000

int pairs[12];
int i, j, popsize, halfpop, sum;
double total;

main()
{
  for (popsize=100; popsize<=2000; popsize+=100) {
    halfpop = popsize>>1;
    pairs[0] = REPS>>1;
    for (i=1; i<12; i++) {
      total = (double)REPS;
      for (j=0; j<i; j++)
        total *= (double)(halfpop-j) / \
                                  (double)((popsize-j)*(j+1));
      total *= ((double)halfpop/ \
```

```
            (double)(popsize-i)+(double)i) / (double)(i+1);
    pairs[i] = (int)(total+0.5);
  }
  sum = -pairs[0];
  for (i=1; i<12; i++)
    sum += pairs[i];
    pairs[0] -= sum;
    printf("popsize = %d\n", popsize);
    printf("pairs[0]:\t%5d\t%5d\n", pairs[0], pairs[0]);
    for (i=1; i<12; i++) {
      printf("pairs[%d]:\t%5d\t", i, pairs[i]);
      pairs[i] += pairs[i-1];
      printf("%5d\n", pairs[i]);
    }
    printf("sum = %d\n\n", sum);
  }
}
```

D.2 Output for pairs.c

```
popsize = 100
pairs[0]:      5000      5000
pairs[1]:      3763      8763
pairs[2]:      1035      9798
pairs[3]:       178      9976
pairs[4]:        22      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = 0

popsize = 200
pairs[0]:      5000      5000
pairs[1]:      3756      8756
pairs[2]:      1039      9795
pairs[3]:       180      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = 0

popsize = 300
pairs[0]:      5000      5000
pairs[1]:      3754      8754
pairs[2]:      1040      9794
pairs[3]:       181      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
```

```
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = 0

popsize = 400
pairs[0]:       5001      5001
pairs[1]:       3753      8754
pairs[2]:       1040      9794
pairs[3]:        181      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 500
pairs[0]:       5001      5001
pairs[1]:       3753      8754
pairs[2]:       1040      9794
pairs[3]:        181      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 600
pairs[0]:       5000      5000
pairs[1]:       3752      8752
pairs[2]:       1041      9793
pairs[3]:        182      9975
```

```
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = 0

popsize = 700
pairs[0]:      5000      5000
pairs[1]:      3752      8752
pairs[2]:      1041      9793
pairs[3]:       182      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = 0

popsize = 800
pairs[0]:      5000      5000
pairs[1]:      3752      8752
pairs[2]:      1041      9793
pairs[3]:       182      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = 0

popsize = 900
```

```
pairs[0]:       5001       5001
pairs[1]:       3751       8752
pairs[2]:       1041       9793
pairs[3]:        182       9975
pairs[4]:         23       9998
pairs[5]:          2      10000
pairs[6]:          0      10000
pairs[7]:          0      10000
pairs[8]:          0      10000
pairs[9]:          0      10000
pairs[10]:         0      10000
pairs[11]:         0      10000
sum = -1

popsize = 1000
pairs[0]:       5001       5001
pairs[1]:       3751       8752
pairs[2]:       1041       9793
pairs[3]:        182       9975
pairs[4]:         23       9998
pairs[5]:          2      10000
pairs[6]:          0      10000
pairs[7]:          0      10000
pairs[8]:          0      10000
pairs[9]:          0      10000
pairs[10]:         0      10000
pairs[11]:         0      10000
sum = -1

popsize = 1100
pairs[0]:       5001       5001
pairs[1]:       3751       8752
pairs[2]:       1041       9793
pairs[3]:        182       9975
pairs[4]:         23       9998
pairs[5]:          2      10000
pairs[6]:          0      10000
pairs[7]:          0      10000
pairs[8]:          0      10000
pairs[9]:          0      10000
pairs[10]:         0      10000
```

```
pairs[11]:        0     10000
sum = -1

popsize = 1200
pairs[0]:      5001      5001
pairs[1]:      3751      8752
pairs[2]:      1041      9793
pairs[3]:       182      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = -1

popsize = 1300
pairs[0]:      5001      5001
pairs[1]:      3751      8752
pairs[2]:      1041      9793
pairs[3]:       182      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = -1

popsize = 1400
pairs[0]:      5001      5001
pairs[1]:      3751      8752
pairs[2]:      1041      9793
pairs[3]:       182      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
```

```
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 1500
pairs[0]:       5001      5001
pairs[1]:       3751      8752
pairs[2]:       1041      9793
pairs[3]:        182      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 1600
pairs[0]:       5001      5001
pairs[1]:       3751      8752
pairs[2]:       1041      9793
pairs[3]:        182      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 1700
pairs[0]:       5001      5001
pairs[1]:       3751      8752
pairs[2]:       1041      9793
```

```
pairs[3]:        182      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 1800
pairs[0]:       5001      5001
pairs[1]:       3751      8752
pairs[2]:       1041      9793
pairs[3]:        182      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1

popsize = 1900
pairs[0]:       5001      5001
pairs[1]:       3751      8752
pairs[2]:       1041      9793
pairs[3]:        182      9975
pairs[4]:         23      9998
pairs[5]:          2     10000
pairs[6]:          0     10000
pairs[7]:          0     10000
pairs[8]:          0     10000
pairs[9]:          0     10000
pairs[10]:         0     10000
pairs[11]:         0     10000
sum = -1
```

```
popsize = 2000
pairs[0]:      5001      5001
pairs[1]:      3751      8752
pairs[2]:      1041      9793
pairs[3]:       182      9975
pairs[4]:        23      9998
pairs[5]:         2     10000
pairs[6]:         0     10000
pairs[7]:         0     10000
pairs[8]:         0     10000
pairs[9]:         0     10000
pairs[10]:        0     10000
pairs[11]:        0     10000
sum = -1
```

Appendix E

Mutation in Dynamic Environments Program: mute

E.1 Source Code for mute.c

```
/*
PROGRAM:   mute.c
AUTHOR:    Tim Watson
MODIFIED: April 2003

Program to test hypothesis that, as the speed of
environmental change increases, there is an associated
selective pressure for higher mutation rates within an
organism.

  o Matrix p holds current/next population, each element in
    p represents a particular genotype consisting of two
    genes: a gene for one of the eight possible mutation
    rates, and a gene for one of the thirty-two possible
    phenotype values.
  o Matrix f holds the fitness function: the first element
    contains the relative fitness of the optimum phenotype
    value, and so on.
  o Matrix m holds the distribution functions of the eight
    possible mutation rates. Each column is a distribution
    (all sum to 32): the first represents no mutation, the
    last uniformly random mutation.
  o The population is initially uniform. Then, for each
```

```
    generation, each row (representing a particular
    phenotype value) is scaled by its relative fitness;
    next, each column has its distribution changed according
    to the mutation distribution, and finally the entire
    population is renormalised to keep the population size
    constant.
  o Output is produced showing the total number of
    individuals for each mutation rate, for each generation.
*/

#include <stdio.h>

double p[2][32][8];
double f[32];
double m[32][8] = {{32, 16, 12, 10, 8.75, 7.875, 7.21875, 1},
                   {0, 8, 8, 7.5, 7, 6.5625, 6.1875, 1},
                   {0, 0, 2, 3, 3.5, 3.75, 3.8671875, 1},
                   {0, 0, 0, 0.5, 1, 1.40625, 1.71875, 1},
                   {0, 0, 0, 0, 0.125, 0.3125, 0.515625, 1},
                   {0, 0, 0, 0, 0, 0.03125, 0.09375, 1},
                   {0, 0, 0, 0, 0, 0, 0.0078125, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0, 1},
                   {0, 0, 0, 0, 0, 0, 0.0078125, 1},
```

```
                                {0, 0, 0, 0, 0, 0.03125, 0.09375, 1},
                                {0, 0, 0, 0, 0.125, 0.3125, 0.515625, 1},
                                {0, 0, 0, 0.5, 1, 1.40625, 1.71875, 1},
                                {0, 0, 2, 3, 3.5, 3.75, 3.8671875, 1},
                                {0, 8, 8, 7.5, 7, 6.5625, 6.1875, 1}};

int next_opt(void);

int main()
{
    int i, j, k, gen;
    double count;

    for (i=0; i<32; i++) {          /* initialise */
        f[i] = m[i][0];
        for (j=0; j<8; j++)
            p[0][i][j] = 8.0;
    }

    for (gen=0; gen<100; gen++) {
        k = ~ next_opt();           /* next gen by fitness */
        for (i=0; i<32; i++) {
            k = ++k & 31;
            for (j=0; j<8; j++)
                p[1][i][j] = p[0][i][j] * f[k];
        }

        for (i=0; i<32; i++)        /* mutational drift */
            for (j=0; j<8; j++) {
                count = 0.0;
                for (k=0; k<32; k++)
                    count += p[1][i+k&31][j] * m[k][j];
                p[0][i][j] = count;
            }

        count = 0.0;                /* renormalise */
        for (i=0; i<32; i++)
            for (j=0; j<8; j++)
                count += p[0][i][j];
        count /= 2048.0;
        for (i=0; i<32; i++)
```

```
        for (j=0; j<8; j++)
            p[0][i][j] /= count;

    for (i=0; i<8; i++) { /* totals for each muterate */
        count = 0.0;
        for (j=0; j<32; j++)
            count += p[0][j][i];
        printf("%.0f\t", count);
    }
    putchar('\n');
}

    return 0;
}

int next_opt()
{
    static int opt;

    return (opt += 2);
}
```

E.2 Output for mute.c: One Step per Generation

```
# f = m[][0], next_opt = opt++
 256   256   256   256   256   256   256   256
   0   370   370   347   324   304   286    46
   0   415   415   365   318   279   248     6
   0   454   454   374   304   251   210     1
   0   491   491   379   288   222   176     0
   0   527   527   382   270   196   146     0
   0   562   562   381   252   171   120     0
   0   595   595   378   233   149    98     0
   0   626   626   373   215   128    80     0
   0   655   655   366   197   110    65     0
   0   682   682   358   179    94    52     0
   0   708   708   348   163    80    42     0
   0   731   731   337   147    68    34     0
   0   753   753   325   133    57    27     0
   0   773   773   313   119    48    21     0
   0   792   792   301   107    41    17     0
   0   809   809   288    95    34    13     0
   0   824   824   275    85    28    10     0
   0   839   839   263    76    24     8     0
   0   852   852   250    67    20     6     0
   0   864   864   238    60    16     5     0
   0   876   876   226    53    14     4     0
   0   886   886   214    47    11     3     0
   0   896   896   203    42     9     2     0
   0   905   905   192    37     8     2     0
   0   913   913   182    32     6     1     0
   0   921   921   172    29     5     1     0
   0   928   928   162    25     4     1     0
   0   934   934   153    22     4     1     0
   0   940   940   145    20     3     1     0
   0   946   946   136    17     2     0     0
   0   951   951   129    15     2     0     0
   0   956   956   121    13     2     0     0
   0   960   960   114    12     1     0     0
   0   964   964   107    10     1     0     0
   0   968   968   101     9     1     0     0
```

0	972	972	95	8	1	0	0
0	975	975	90	7	1	0	0
0	979	979	84	6	1	0	0
0	981	981	79	5	0	0	0
0	984	984	74	5	0	0	0
0	987	987	70	4	0	0	0
0	989	989	66	4	0	0	0
0	991	991	62	3	0	0	0
0	993	993	58	3	0	0	0
0	995	995	55	2	0	0	0
0	997	997	51	2	0	0	0
0	999	999	48	2	0	0	0
0	1001	1001	45	2	0	0	0
0	1002	1002	42	1	0	0	0
0	1003	1003	40	1	0	0	0
0	1005	1005	37	1	0	0	0
0	1006	1006	35	1	0	0	0
0	1007	1007	33	1	0	0	0
0	1008	1008	31	1	0	0	0
0	1009	1009	29	1	0	0	0
0	1010	1010	27	1	0	0	0
0	1011	1011	26	1	0	0	0
0	1012	1012	24	0	0	0	0
0	1013	1013	22	0	0	0	0
0	1013	1013	21	0	0	0	0
0	1014	1014	20	0	0	0	0
0	1015	1015	19	0	0	0	0
0	1015	1015	17	0	0	0	0
0	1016	1016	16	0	0	0	0
0	1016	1016	15	0	0	0	0
0	1017	1017	14	0	0	0	0
0	1017	1017	13	0	0	0	0
0	1018	1018	13	0	0	0	0
0	1018	1018	12	0	0	0	0
0	1018	1018	11	0	0	0	0
0	1019	1019	10	0	0	0	0
0	1019	1019	10	0	0	0	0
0	1019	1019	9	0	0	0	0
0	1020	1020	9	0	0	0	0
0	1020	1020	8	0	0	0	0
0	1020	1020	8	0	0	0	0

0	1020	1020	7	0	0	0	0
0	1021	1021	7	0	0	0	0
0	1021	1021	6	0	0	0	0
0	1021	1021	6	0	0	0	0
0	1021	1021	5	0	0	0	0
0	1021	1021	5	0	0	0	0
0	1022	1022	5	0	0	0	0
0	1022	1022	4	0	0	0	0
0	1022	1022	4	0	0	0	0
0	1022	1022	4	0	0	0	0
0	1022	1022	3	0	0	0	0
0	1022	1022	3	0	0	0	0
0	1022	1022	3	0	0	0	0
0	1023	1023	3	0	0	0	0
0	1023	1023	3	0	0	0	0
0	1023	1023	3	0	0	0	0
0	1023	1023	2	0	0	0	0
0	1023	1023	2	0	0	0	0
0	1023	1023	2	0	0	0	0
0	1023	1023	2	0	0	0	0
0	1023	1023	2	0	0	0	0

E.3 Output for mute.c: Two Steps per Generation

```
# f = m[][0], next_opt = opt + 2
 256   256   256   256   256   256   256   256
   0     0   239   359   419   449   463   120
   0     0   148   334   454   521   554    37
   0     0    86   292   464   570   625    11
   0     0    49   248   459   605   684     3
   0     0    27   207   447   631   736     1
   0     0    15   170   430   650   782     0
   0     0     8   139   410   665   825     0
   0     0     4   113   389   676   865     0
   0     0     2    92   368   684   902     0
   0     0     1    74   346   689   938     0
   0     0     1    59   324   692   971     0
   0     0     0    48   303   694  1003     0
   0     0     0    38   283   693  1034     0
   0     0     0    30   263   691  1063     0
   0     0     0    24   244   688  1091     0
   0     0     0    19   227   684  1118     0
   0     0     0    15   210   678  1144     0
   0     0     0    12   194   672  1170     0
   0     0     0    10   179   665  1194     0
   0     0     0     8   166   658  1217     0
   0     0     0     6   153   650  1240     0
   0     0     0     5   141   641  1262     0
   0     0     0     4   129   632  1283     0
   0     0     0     3   119   623  1303     0
   0     0     0     2   109   613  1323     0
   0     0     0     2   100   603  1343     0
   0     0     0     1    92   593  1361     0
   0     0     0     1    84   583  1380     0
   0     0     0     1    77   572  1397     0
   0     0     0     1    71   562  1414     0
   0     0     0     1    65   551  1431     0
   0     0     0     0    59   541  1447     0
   0     0     0     0    54   530  1463     0
   0     0     0     0    50   519  1479     0
   0     0     0     0    45   509  1494     0
```

0	0	0	0	42	498	1508	0
0	0	0	0	38	488	1522	0
0	0	0	0	35	477	1536	0
0	0	0	0	32	467	1550	0
0	0	0	0	29	456	1563	0
0	0	0	0	26	446	1575	0
0	0	0	0	24	436	1588	0
0	0	0	0	22	426	1600	0
0	0	0	0	20	416	1612	0
0	0	0	0	18	406	1623	0
0	0	0	0	17	397	1635	0
0	0	0	0	15	387	1645	0
0	0	0	0	14	378	1656	0
0	0	0	0	13	369	1666	0
0	0	0	0	11	360	1677	0
0	0	0	0	10	351	1686	0
0	0	0	0	9	342	1696	0
0	0	0	0	9	334	1706	0
0	0	0	0	8	325	1715	0
0	0	0	0	7	317	1724	0
0	0	0	0	6	309	1732	0
0	0	0	0	6	301	1741	0
0	0	0	0	5	294	1749	0
0	0	0	0	5	286	1757	0
0	0	0	0	4	279	1765	0
0	0	0	0	4	271	1773	0
0	0	0	0	4	264	1780	0
0	0	0	0	3	257	1787	0
0	0	0	0	3	250	1795	0
0	0	0	0	3	244	1801	0
0	0	0	0	2	237	1808	0
0	0	0	0	2	231	1815	0
0	0	0	0	2	225	1821	0
0	0	0	0	2	219	1827	0
0	0	0	0	2	213	1834	0
0	0	0	0	2	207	1840	0
0	0	0	0	1	201	1845	0
0	0	0	0	1	196	1851	0
0	0	0	0	1	190	1856	0
0	0	0	0	1	185	1862	0
0	0	0	0	1	180	1867	0

0	0	0	0	1	175	1872	0
0	0	0	0	1	170	1877	0
0	0	0	0	1	166	1882	0
0	0	0	0	1	161	1886	0
0	0	0	0	1	156	1891	0
0	0	0	0	1	152	1895	0
0	0	0	0	0	148	1900	0
0	0	0	0	0	144	1904	0
0	0	0	0	0	140	1908	0
0	0	0	0	0	136	1912	0
0	0	0	0	0	132	1916	0
0	0	0	0	0	128	1920	0
0	0	0	0	0	124	1923	0
0	0	0	0	0	121	1927	0
0	0	0	0	0	117	1930	0
0	0	0	0	0	114	1934	0
0	0	0	0	0	111	1937	0
0	0	0	0	0	108	1940	0
0	0	0	0	0	104	1943	0
0	0	0	0	0	101	1946	0
0	0	0	0	0	99	1949	0
0	0	0	0	0	96	1952	0
0	0	0	0	0	93	1955	0

Appendix F

Automatic Mutation Rate Program: automute

F.1 Source Code for automute.c

```
/*
PROG:      automute.c
AUTHOR:    Tim Watson
MODIFIED: April 2003

Test of auto-mutation gene for rapid adaptation to
discontinuous change.
*/



#include <stdio.h>
#include "r250.h"
#include "randlcg.h"



#define POPMAX      1000
#define CHROMAX       80


unsigned char curr[POPMAX][CHROMAX/8], next[POPMAX][CHROMAX/8];
```

```
double fit[POPMAX];
int popsize, chromsize, crossrate, muterate,
    runmax, genmax, muteboost, fitswap, seed;


void init_var(void);
void display_var(void);
void init_pop(void);
void calc_fit(int);
void display_stats(int);
void next_gen(void);



main()
{
  int run, gen;

  init_var();
  display_var();
  r250_init(seed);

  for (run=0; run<runmax || runmax==0; run++) {
    init_pop();

    for (gen=0; gen<genmax || genmax==0; gen++) {
      calc_fit(gen);
      display_stats(gen);
      if (gen<genmax-1 || genmax==0)
        next_gen();
    }
  }
  return 0;
}
```

```
void init_var()
{
  FILE *setup;
```

```
if (setup = fopen("automute.dat", "r")) {
  fscanf(setup, "popsize   %u \
         chromsize %u \
         crossrate %u \
         muterate  %u \
         runs      %u \
         gens      %u \
         muteboost %u \
         fitswap   %u \
         seed      %u", &popsize, &chromsize, &crossrate, \
                        &muterate, &runmax, &genmax, \
                        &muteboost, &fitswap, &seed);
    fclose(setup);
  }
  else {
    fprintf(stderr, "automute: can't open file automute.dat");
    exit(1);
  }
}
```

```
void display_var()
{
  printf("popsize   %u\tchromsize %u \
      \ncrossrate %u\tmuterate  %u \
      \nmuteboost %u\tfitswap   %u \
      \nseed      %u\n\n", popsize, chromsize, crossrate,
                    muterate, muteboost, fitswap, seed);
}
```

```
void init_pop()
{
  int i, j;

  for (i=0; i<popsize; i++)
    for (j=0; j<chromsize/8; j++)
      curr[i][j] = r250() % 256;
}
```

```
void calc_fit(int gen)
{
  int i, j;
  unsigned char mask;

if (gen / fitswap % 2)
  mask = 0;
else
  mask = 127;

for (i=0; i<popsize; i++) {
  if ((mask<<1) ^ (curr[i][0]<<1))
    fit[i] = 1.0;
  else {
    j = 1;
    while (j<chromsize/8 && !(mask^curr[i][j]))
      j++;
    if (j < chromsize/8)
      fit[i] = 1.0;
    else
      fit[i] = 10.0;
  }
  }
}



void display_stats(int gen)
{
  int i, count;
  double av_fit, high_mute;

av_fit = 0.0;
count = 0;
for (i=0; i<popsize; i++) {
  av_fit += fit[i];
  if (curr[i][0] & 1<<7)
    count++;
```

```
  }

av_fit /= (double) popsize;
high_mute = (double) count * 100 / (double) popsize;

printf("gen %5d\tmean fit = %5.2f\t%% high mute = %5.2f\n",
        gen, av_fit, high_mute);
}



void next_gen()
{
  int i, j, k, mute;
  unsigned char cut, mask;
  double choice;

  for (i=1; i<popsize; i++)
    fit[i] += fit[i-1];

  for (i=0; i<popsize; i++) { /* produce next generation */
    choice = fit[popsize-1] * dr250();
    j = 0;
    while ((fit[j] <= choice) && (j < popsize-1))
      j++;
    for (k=0; k<chromsize/8; k++)
      next[i][k] = curr[j][k];
}

if (crossrate)                                  /* crossover */
  for (i=0; i<popsize-1; i+=2)
    if (r250() % 100 < crossrate) {
      cut = r250() % (chromsize - 1) + 1;
      for (j=0; j<cut/8; j++) {
        curr[i][j]   = next[i][j];
    curr[i+1][j] = next[i+1][j];
}
if (cut % 8) {
  mask = 255 >> cut % 8;
  curr[i][j]   = (next[i][j] & ~mask) + \
                 (next[i+1][j] & mask);
```

```
    curr[i+1][j] = (next[i+1][j] & ~mask) + \
                   (next[i][j] & mask);
    j++;
  }
  for (; j<chromsize/8; j++) {
    curr[i][j]   = next[i+1][j];
    curr[i+1][j] = next[i][j];
    }
  }
  else
    for (j=0; j<chromsize/8; j++) {
      curr[i][j]   = next[i][j];
      curr[i+1][j] = next[i+1][j];
    }
else                                        /* no crossover */
  for (i=0; i<popsize; i++)
    for (j=0; j<chromsize/8; j++)
      curr[i][j] = next[i][j];

if (muterate)                                /* mutation */
  for (i=0; i<popsize; i++) {
    if (curr[i][0] & ~0<<7)
      mute = muterate / muteboost;
    else
      mute = muterate;

    for (j=0; j<chromsize; j++)
      if (r250() % mute == 0)
        curr[i][j/8] ^= 1 << 7 - j % 8;
  }
}
```

F.2 Sample Data File: automute.dat

```
popsize   200
chromsize  16
crossrate  70
muterate  500
runs       30
gens     6000
muteboost   5
fitswap  3000
seed       10

-------------------------------------------------------
popsize:   multiple of 10.
chromsize: multiple of 8.
crossrate: percentage of pairs in next generation that have
           one point crossover applied to them.
muterate:  one in muterate chance that a bit will mutate.
runs:      number of test runs (0 = infinite).
gens:      number of generations (0 = infinite).
muteboost: high muterate = muterate / muteboost.
fitswap:   toggle fitness optimum every fitswap generations.
seed:      initial random number seed (must be > 0).
-------------------------------------------------------
```

F.3 Output for automute.c (Condensed Version)

```
gen    100 mean fit =  9.78 % high mute =  0.10
gen    200 mean fit =  9.79 % high mute =  0.20
gen    300 mean fit =  9.78 % high mute =  0.00
gen    400 mean fit =  9.88 % high mute =  0.10
gen    500 mean fit =  9.88 % high mute =  0.00
gen    600 mean fit =  9.87 % high mute =  0.00
gen    700 mean fit =  9.92 % high mute =  0.00
gen    800 mean fit =  9.87 % high mute =  0.00
gen    900 mean fit =  9.86 % high mute =  0.00
gen   1000 mean fit =  9.88 % high mute =  0.00
gen   1100 mean fit =  9.86 % high mute =  0.00
gen   1200 mean fit =  9.89 % high mute =  0.00
gen   1300 mean fit =  9.86 % high mute =  0.10
gen   1400 mean fit =  9.88 % high mute =  0.00
gen   1500 mean fit =  9.85 % high mute =  0.20
gen   1600 mean fit =  9.84 % high mute =  0.00
gen   1700 mean fit =  9.86 % high mute =  0.10
gen   1800 mean fit =  9.81 % high mute =  0.30
gen   1900 mean fit =  9.87 % high mute =  0.10
gen   2000 mean fit =  1.00 % high mute =  0.10
gen   2100 mean fit =  1.00 % high mute = 26.20
gen   2200 mean fit =  1.00 % high mute = 22.00
gen   2300 mean fit =  1.00 % high mute = 67.60
gen   2400 mean fit =  1.00 % high mute = 26.40
gen   2500 mean fit =  1.00 % high mute = 27.40
gen   2600 mean fit =  1.00 % high mute = 21.60
gen   2700 mean fit =  1.00 % high mute = 13.60
gen   2800 mean fit =  1.00 % high mute =  4.80
gen   2900 mean fit =  1.00 % high mute = 18.00
gen   3000 mean fit =  1.00 % high mute = 13.30
gen   3100 mean fit =  9.87 % high mute =  0.00
gen   3200 mean fit =  9.86 % high mute =  0.10
gen   3300 mean fit =  9.83 % high mute =  0.10
gen   3400 mean fit =  9.91 % high mute =  0.00
gen   3500 mean fit =  9.81 % high mute =  0.20
gen   3600 mean fit =  9.74 % high mute =  0.10
gen   3700 mean fit =  9.76 % high mute =  0.20
```

```
gen  3800 mean fit =  9.84 % high mute =  0.00
gen  3900 mean fit =  9.78 % high mute =  0.00
gen  4000 mean fit =  1.00 % high mute =  0.00
gen  4100 mean fit =  1.00 % high mute =  1.10
gen  4200 mean fit =  1.00 % high mute = 12.20
gen  4300 mean fit =  1.00 % high mute = 10.40
gen  4400 mean fit =  1.00 % high mute = 20.40
gen  4500 mean fit =  1.00 % high mute = 25.60
gen  4600 mean fit =  1.00 % high mute = 32.40
gen  4700 mean fit =  1.00 % high mute = 29.50
gen  4800 mean fit =  1.00 % high mute = 39.20
gen  4900 mean fit =  1.00 % high mute = 26.30
gen  5000 mean fit =  1.00 % high mute = 29.10
gen  5100 mean fit =  1.00 % high mute = 20.40
gen  5200 mean fit =  1.00 % high mute =  8.20
gen  5300 mean fit =  9.83 % high mute =  0.20
gen  5400 mean fit =  9.84 % high mute =  0.00
gen  5500 mean fit =  9.82 % high mute =  0.10
gen  5600 mean fit =  9.89 % high mute =  0.20
gen  5700 mean fit =  9.80 % high mute =  0.10
gen  5800 mean fit =  9.82 % high mute =  0.10
gen  5900 mean fit =  9.78 % high mute =  0.20
gen  6000 mean fit =  1.00 % high mute =  0.50
gen  6100 mean fit =  1.00 % high mute = 14.30
gen  6200 mean fit =  1.00 % high mute = 21.60
gen  6300 mean fit =  1.00 % high mute = 35.60
gen  6400 mean fit =  1.00 % high mute = 66.20
gen  6500 mean fit =  1.00 % high mute = 49.50
gen  6600 mean fit =  1.00 % high mute = 35.40
gen  6700 mean fit =  9.89 % high mute =  0.00
gen  6800 mean fit =  9.84 % high mute =  0.40
gen  6900 mean fit =  9.80 % high mute =  0.20
gen  7000 mean fit =  9.84 % high mute =  0.00
gen  7100 mean fit =  9.93 % high mute =  0.10
gen  7200 mean fit =  9.80 % high mute =  0.20
gen  7300 mean fit =  9.84 % high mute =  0.20
gen  7400 mean fit =  9.81 % high mute =  0.20
gen  7500 mean fit =  9.84 % high mute =  0.20
gen  7600 mean fit =  9.84 % high mute =  0.20
gen  7700 mean fit =  9.85 % high mute =  0.10
gen  7800 mean fit =  9.88 % high mute =  0.00
```

```
gen  7900 mean fit =  9.76 % high mute =  0.40
gen  8000 mean fit =  1.00 % high mute =  0.10
gen  8100 mean fit =  1.00 % high mute = 10.30
gen  8200 mean fit =  1.00 % high mute = 15.70
gen  8300 mean fit =  1.00 % high mute =  9.20
gen  8400 mean fit =  1.00 % high mute = 26.50
gen  8500 mean fit =  1.00 % high mute =  6.60
gen  8600 mean fit =  1.00 % high mute = 12.40
gen  8700 mean fit =  1.00 % high mute = 28.20
gen  8800 mean fit =  1.00 % high mute = 23.10
gen  8900 mean fit =  9.87 % high mute =  0.00
gen  9000 mean fit =  9.87 % high mute =  0.10
gen  9100 mean fit =  9.86 % high mute =  0.10
gen  9200 mean fit =  9.85 % high mute =  0.30
gen  9300 mean fit =  9.78 % high mute =  0.10
gen  9400 mean fit =  9.85 % high mute =  0.00
gen  9500 mean fit =  9.82 % high mute =  0.10
gen  9600 mean fit =  9.87 % high mute =  0.20
gen  9700 mean fit =  9.86 % high mute =  0.00
gen  9800 mean fit =  9.81 % high mute =  0.00
gen  9900 mean fit =  9.84 % high mute =  0.00
```

Appendix G

Automatic Hamming Distance Program: autoham

G.1 Source Code for autoham.c

```
/*
PROGRAM:   autoham.c
AUTHOR:    Tim Watson
MODIFIED: April 2003

A program to test the relative abilities of a simple GA,
an automute GA,  an autoham GA and a combined automute
and autoham GA - with and without a complementary member
in each generation - to keep track of a discontinuously
changing environment.  A flat fitness function is used,
with a single,  spike optimum,  which is toggled whenever
the population reaches 95% of the optimal population
fitness. Statistics are generated for each type of GA
based on the number of generations taken to find the
optimum after it is toggled.
*/

#include <stdio.h>
#include <math.h>
#include "r250.h"
#include "randlcg.h"

#define POPSIZE          300     /* divisible by 4        */
```

```
#define CROSS_RATE      70      /* 70% crossover        */
#define MUTE_RATE      500      /* 1 in 500 bits mutate */
#define MUTE_BOOST      20      /* divisor of MUTE_RATE */
#define OPT_FITNESS     20
#define SWAP_THRESH     19      /* 95% optimal fitness  */
#define GEN_LIMIT     4000      /* max gens before swap */
#define RUNS           100
#define RAND_SEED        3
#define HAM_PROB         5      /* prob of ham fitness  */

unsigned char curr[POPSIZE][2], next[POPSIZE][2];
unsigned char opt[2][2] = {{32,36}, {106,188}};
int fit[POPSIZE], result[RUNS];
double mean, stdev;

void display_exp_header(int, int);
void init_pop(int);
void calc_fit(int);
void next_gen(int, int);
void swap_opt(void);
void display_result(int, int);
void calc_statistics(void);
void display_statistics(void);



main()
{
  int compl, exp, swap, gen;

  r250_init(RAND_SEED);
  for (compl=0; compl<2; compl++)
    for (exp=0; exp<5; exp++) {
      display_exp_header(exp,compl);
      init_pop(compl);
gen = 0;
calc_fit(exp);
while (fit[POPSIZE-1] < SWAP_THRESH*POPSIZE) {
  next_gen(exp,compl);
  calc_fit(exp);
  if (gen < GEN_LIMIT)
```

```
      gen++;
    else {
      printf("\nDidn't find first optimum in time...");
      break;
    }
  }
  for (swap=0; swap<RUNS; swap++) {
    swap_opt();
    gen = 0;
    calc_fit(exp);
    while (fit[POPSIZE-1] < SWAP_THRESH*POPSIZE) {
      next_gen(exp,compl);
      calc_fit(exp);
      if (gen < GEN_LIMIT)
        gen++;
      else
        break;
    }
    result[swap] = gen;
    display_result(gen,swap);
  }
    calc_statistics();
    display_statistics();
  }
  return 0;
}



void display_exp_header(int exp, int compl)
{
  switch (exp) {
  case 0:  printf("Simple GA");
           break;
  case 1:  printf("Automute GA");
           break;
  case 2:  printf("Autoham GA");
           break;
  case 3:  printf("Automute and autoham GA, two control bits");
           break;
  case 4:  printf("Automute and autoham GA, one control bit");
```

```
            break;
default:    printf("Undefined experiment");
            break;
}
if (compl)
  printf(", with complement:");
else
    printf(":");
}



void init_pop(int compl)
{
  int i, j;

  for (i=0; i<POPSIZE; i++)
    for (j=0; j<2; j++)
      curr[i][j] = 0;
  if (compl) {
    curr[POPSIZE-1][0] = 255;
    curr[POPSIZE-1][1] = 252;
  }
}



void calc_fit(int exp)
{
  int i, j, xor;

  fit[0] = 1;
  if (exp>1 && curr[0][0]&128 && r250()%HAM_PROB==0)
    for (j=0; j<2; j++)
      for (xor=curr[0][j]^curr[POPSIZE-1][j]; xor!=0; xor>>=1)
        fit[0] += xor % 2;
else if ((curr[0][0]&126)==opt[0][0]&&curr[0][1]==opt[0][1])
  fit[0] = OPT_FITNESS;
for (i=1; i<POPSIZE; i++) {
  fit[i] = fit[i-1] +1;
  if (exp>1 && curr[i][0]&128 && r250()%HAM_PROB==0)
```

```
    for (j=0; j<2; j++)
      for (xor=curr[i][j]^curr[i-1][j]; xor!=0; xor>>=1)
        fit[i] += xor % 2;
  else if ((curr[i][0]&126)==opt[0][0] && \
                                        curr[i][1]==opt[0][1])
    fit[i] = fit[i-1] + OPT_FITNESS;
}
```

```
void next_gen(int exp, int compl)
{
  int i, j, a, b, choice, mask, mute, source, dest;

  for (i=0; i<POPSIZE; i++) {                    /* selection */
    choice = (r250() % fit[POPSIZE-1]) + 1;
    j=0;
    while (choice > fit[j])
        j++;
    next[i][0] = curr[j][0];
    next[i][1] = curr[j][1];
  }

  for (i=0; i<POPSIZE; i+=2)                     /* crossover */
    if (r250()%100 < CROSS_RATE) {
      if (i/2 % 2) {
        a = i - 1;
        b = i + 1;
      }
      else {
        a = i;
        b = i +2;
      }
      choice = r250() % 14;
  if (choice < 7) {
    mask = (next[i][0]^next[i+1][0]) & 127>>choice;
    curr[a][0] = next[i][0] ^ mask;
    curr[b][0] = next[i+1][0] ^ mask;
    curr[a][1] = next[i+1][1];
    curr[b][1] = next [i][1];
```

```
}
else if (choice > 7) {
  curr[a][0] = next[i][0];
  curr[b][0] = next[i+1][0];
  mask = (next[i][1]^next[i+1][1]) & 127>>choice-8;
  curr[a][1] = next[i][1] ^ mask;
  curr[b][1] = next [i+1][1] ^ mask;
  }
  else {
    curr[a][0] = next[i][0];
    curr[b][0] = next[i+1][0];
    curr[a][1] = next[i+1][1];
    curr[b][1] = next [i][1];
  }
 }

if (exp==0 || exp==2)                    /* mutation */
  for (i=0; i<POPSIZE; i++) {
    for (j=0; j<8; j++)
      if (r250()%MUTE_RATE == 0)
        curr[i][0] ^= 128 >> j;
    for (j=0; j<6; j++)
      if (r250()%MUTE_RATE == 0)
        curr[i][1] ^= 128 >> j;
  }
else if (exp == 4) /* automute controlled by leftmost bit */
  for (i=0; i<POPSIZE; i++) {
    mute = MUTE_RATE;
    if (curr[i][0] & 128)
      mute /= MUTE_BOOST;
    for (j=0; j<8; j++)
      if (r250()%mute == 0)
        curr[i][0] ^= 128 >> j;
    for (j=0; j<6; j++)
      if (r250()%mute == 0)
        curr[i][1] ^= 128 >> j;
  }
else                    /* automute controlled by rightmost bit */
  for (i=0; i<POPSIZE; i++) {
    mute = MUTE_RATE;
    if (curr[i][0] & 1)
```

```
    mute /= MUTE_BOOST;
  for (j=0; j<8; j++)
    if (r250()%mute == 0)
      curr[i][0] ^= 128 >> j;
  for (j=0; j<6; j++)
    if (r250()%mute == 0)
      curr[i][1] ^= 128 >> j;
    }

  if (compl) {                          /* complement */
    source = r250() % POPSIZE;
    while (source == (dest=r250()%POPSIZE))
      ;
    curr[dest][0] = curr[source][0] ^ 255;
    curr[dest][1] = curr[source][1] ^ 252;
  }
}
```

```
void swap_opt()
{
  int i;
  unsigned char temp;

  for (i=0; i<2; i++) {
    temp = opt[0][i];
    opt[0][i] = opt[1][i];
    opt[1][i] = temp;
  }
}
```

```
void display_result(int gen, int swap)
{
  if (swap%10)
    printf("%6d",gen);
  else
    printf("\n%6d",gen);
}
```

```
void calc_statistics()
{
  int i;
  double diff;

  mean = stdev = 0.0;
  for (i=0; i<RUNS; i++)
    mean += (double) result[i] / RUNS;
  for (i=0; i<RUNS; i++) {
    diff = mean - result[i];
    stdev += diff * diff / RUNS;
  }
  stdev = sqrt(stdev);
}
```

```
void display_statistics()
{
  printf("\nMean = %6.2f\tStandard Deviation = %6.2f\n\n",
         mean,stdev);
}
```

G.2 Output for autoham.c

```
Simple GA:
  2564   177  2075   579   600   224   290   527   230  1880
   267   370   236   172     7  1082   929   239   715   544
   669   406   395   456   113   251   449   127   981   609
   456   346   991   157   203   416   266   428   376  1244
   162   144   289   805   452   469   352   250     8     6
   160   472   616   424   227   310  1110   549   344   357
   822  1162   168   397   624   494   844   326   450   515
   923   424   531   175   351   441   380   195   191   148
   437   336   162   178   724   359   352   404   197   276
   444   478   629   256   304   550   280   285  1410   195
Mean = 483.69 Standard Deviation = 406.68

Automute GA:
   227   154   270   197   102   328   671   116   241    96
   125   222   716   570   150   226   426   276   192   101
   294   209   339   142   377   723   113   232   233   355
   172   311   312   190   179   342   213   139   443   106
   112   189   161   137   122   149   147   191   115    92
   475   315   216   422   498   140   408   171   374   128
   176    91   321   240   229   495   133   208    81   164
   355    89   228   200   540    84   244   533   137   214
   356   260   184    48   130   447   106   133   134   148
   212   212   216   196   252   133   112    92   110   145
Mean = 237.80 Standard Deviation = 143.16

Autoham GA:
   197   172   126   121   218   162   117   119   163   150
   148   227   250   228   123   143   210   329   162   476
   122   239   230   251   195   295   150    96    99   122
   171    92   248   193   133   104   216   168    77   149
   186   165   126    84   440   147    93   128   268   120
   108   105   171   138   113   118   169   181   139   194
   198   200   160   131   272   158   220   260   251   125
   107   149   263   356   240   120   152    75   333   143
   369   294   132   365   168   169   135   120   184   154
   134   178   179   141   179   335   187   121   143   270
Mean = 182.54 Standard Deviation =  77.26
```

Automute and autoham GA, two control bits:

174	138	158	143	111	147	129	113	148	83
61	113	169	204	146	127	227	185	173	92
129	131	162	153	140	109	128	75	131	190
287	117	132	149	126	87	95	136	155	199
93	87	120	176	131	121	169	98	146	213
154	201	93	132	86	120	80	102	67	302
47	70	108	93	111	65	186	85	198	102
183	85	136	121	104	134	98	136	63	90
80	88	88	86	109	103	61	274	216	172
155	120	134	303	107	111	102	104	197	215

Mean = 134.03 Standard Deviation = 51.49

Automute and autoham GA, one control bit:

232	103	72	160	246	78	47	63	63	155
77	97	103	71	256	58	114	101	101	146
37	102	42	51	60	114	178	48	60	84
53	187	67	61	53	82	46	40	44	90
58	120	35	81	69	40	43	108	44	153
68	77	61	75	71	31	96	85	50	130
68	74	80	41	39	142	34	54	93	61
62	100	37	63	123	58	147	44	138	92
46	50	88	80	40	53	44	46	45	172
38	53	72	23	49	51	88	59	108	54

Mean = 81.46 Standard Deviation = 45.92

Simple GA, with complement:

413	62	93	105	148	165	130	179	145	167
116	148	141	148	217	145	35	217	390	79
169	258	190	97	441	374	383	243	78	97
240	118	243	243	164	133	99	201	339	217
308	95	389	89	231	136	159	116	112	220
206	195	93	577	144	33	232	259	110	283
53	325	81	112	221	212	135	253	115	656
107	447	132	188	315	254	204	398	127	139
169	246	178	58	123	87	127	297	166	79
133	161	74	122	83	138	130	94	187	83

Mean = 187.66 Standard Deviation = 112.18

Automute GA, with complement:

195	129	100	115	80	92	147	115	117	117

70	231	138	111	134	207	55	116	62	323
151	103	90	129	134	67	119	248	125	408
67	179	125	149	101	61	85	41	208	145
92	162	407	53	166	196	280	129	52	141
198	103	109	193	113	130	59	85	142	69
111	144	121	103	191	86	86	73	98	83
124	50	87	109	224	144	207	206	189	94
99	67	224	164	111	76	91	222	55	98
78	266	143	193	60	153	103	126	150	390

Mean = 136.67 Standard Deviation = 72.44

Autoham GA, with complement:

164	127	82	135	69	138	268	112	127	196
69	89	71	86	327	158	214	116	113	75
209	324	103	205	113	112	84	74	129	89
128	99	126	100	137	119	88	263	146	138
136	130	127	82	127	128	158	203	122	94
121	89	101	124	192	104	135	131	99	104
88	119	157	276	99	98	119	310	103	78
103	77	304	306	138	119	227	99	157	99
128	123	204	127	130	209	232	171	76	544
125	145	149	75	424	174	166	102	321	86

Mean = 148.16 Standard Deviation = 79.17

Automute and autoham GA, two control bits, with complement:

103	84	192	145	159	112	157	119	162	126
103	163	316	212	175	78	138	190	102	108
65	166	90	148	101	125	121	114	63	406
250	319	307	105	69	386	95	83	99	50
89	171	143	96	120	199	82	107	91	49
196	179	86	164	93	259	120	126	129	168
152	48	124	81	124	219	60	114	171	317
104	75	195	98	66	69	178	283	179	77
88	201	103	96	88	84	105	59	121	98
125	173	94	152	130	115	192	129	76	105

Mean = 138.41 Standard Deviation = 70.34

Automute and autoham GA, one control bit, with complement:

59	81	28	80	35	42	56	40	39	115
37	62	48	57	48	60	65	30	47	36
54	68	41	86	36	61	60	108	49	57

74	80	135	63	129	46	39	52	170	88
31	43	59	59	44	68	96	85	41	66
69	50	75	83	83	52	39	72	73	218
81	75	87	52	57	29	49	80	50	138
116	70	39	36	74	50	63	92	113	39
53	210	47	89	54	177	86	60	26	139
39	30	79	44	51	67	63	46	339	40

Mean = 70.96 Standard Deviation = 44.81

Appendix H

Pseudo-Random Number Generator

H.1 Source Code for r250.h

```
/* r250.h    prototypes for r250 random number generator,

             Kirkpatrick, S., and E. Stoll, 1981; "A Very Fast
             Shift-Register Sequence Random Number Generator",
             Journal of Computational Physics, V.40

             also:

             see W.L. Maier, DDJ May 1991


*/

#ifndef _R250_H_
#define _R250_H_ 1.2

#ifdef __cplusplus
extern "C" {
#endif

#ifdef NO_PROTO
void          r250_init();
unsigned int r250();
```

```
double      dr250();

#else
void          r250_init(int seed);
unsigned int r250( void );
double       dr250( void );
#endif

#ifdef __cplusplus
}
#endif

#endif
```

H.2 Source Code for r250.c

```
/* r250.c        the r250 uniform random number algorithm

                Kirkpatrick, S., and E. Stoll, 1981; "A Very Fast
                Shift-Register Sequence Random Number Generator",
                Journal of Computational Physics, V.40

                also:

                see W.L. Maier, DDJ May 1991


*/

static char rcsid[] = \
                "@(#)r250.c     1.2 15:50:31 11/21/94    EFC";


#include <limits.h>

#include "r250.h"

/* set the following if you trust rand(), otherwise the
   minimal standard generator is used
*/
/* #define TRUST_RAND */


#ifndef TRUST_RAND
#include "randlcg.h"
#endif

/* defines to allow for 16 or 32 bit integers */
#define BITS 32
/* #define MAIN */

#if WORD_BIT == 32
#ifndef BITS
#define BITS    32
```

```
#endif
#else
#ifndef BITS
#define BITS    16
#endif
#endif

#if BITS == 31
#define MSB             0x40000000L
#define ALL_BITS        0x7fffffffL
#define HALF_RANGE      0x20000000L
#define STEP            7
#endif

#if BITS == 32
#define MSB             0x80000000L
#define ALL_BITS        0xffffffffL
#define HALF_RANGE      0x40000000L
#define STEP            7
#endif

#if BITS == 16
#define MSB             0x8000
#define ALL_BITS        0xffff
#define HALF_RANGE      0x4000
#define STEP            11
#endif

static unsigned int r250_buffer[ 250 ];
static int r250_index;

#ifdef NO_PROTO
void r250_init(sd)
int seed;
#else
void r250_init(int sd)
#endif
{
    int j, k;
    unsigned int mask, msb;
```

```
#ifdef TRUST_RAND

#if BITS == 32 || BITS == 31
    srand48( sd );
#else
    srand( sd );
#endif
```

```
#else
    set_seed( sd );
#endif

    r250_index = 0;
    /* fill r250 buffer with BITS-1 bit values */
    for (j = 0; j < 250; j++)
#ifdef TRUST_RAND
#if BITS == 32 || BITS == 31
        r250_buffer[j] = (unsigned int)lrand48();
#else
        r250_buffer[j] = rand();
#endif
#else
        r250_buffer[j] = randlcg();
#endif


    for (j = 0; j < 250; j++)        /* set some MSBs to 1 */
#ifdef TRUST_RAND
        if ( rand() > HALF_RANGE )
            r250_buffer[j] |= MSB;
#else
        if ( randlcg() > HALF_RANGE )
            r250_buffer[j] |= MSB;
#endif

    msb = MSB;                       /* turn on diagonal bit */
    mask = ALL_BITS;                 /* turn off the leftmost bits */

    for (j = 0; j < BITS; j++)
```

```
{
    k = STEP * j + 3;   /* select a word to operate on */
    r250_buffer[k] &= mask; /*turn off bits left of diag*/
    r250_buffer[k] |= msb;  /* turn on the diagonal bit */
    mask >>= 1;
    msb  >>= 1;
}

}

unsigned int r250()   /* returns a random unsigned integer */
{
    register int    j;
    register unsigned int new_rand;

    if ( r250_index >= 147 )
        j = r250_index - 147;   /* wrap pointer around */
    else
        j = r250_index + 103;

    new_rand = r250_buffer[ r250_index ] ^ r250_buffer[ j ];
    r250_buffer[ r250_index ] = new_rand;

    if ( r250_index >= 249 ) /* inc pointer for next time */
        r250_index = 0;
    else
        r250_index++;

    return new_rand;

}


double dr250()   /* returns a random double in range 0..1 */
{
    register int    j;
    register unsigned int new_rand;

    if ( r250_index >= 147 )
        j = r250_index - 147;   /* wrap pointer around */
    else
```

```
    j = r250_index + 103;

new_rand = r250_buffer[ r250_index ] ^ r250_buffer[ j ];
r250_buffer[ r250_index ] = new_rand;

if ( r250_index >= 249 ) /* inc pointer for next time */
    r250_index = 0;
    else
        r250_index++;

    return (double)new_rand / ALL_BITS;

}

#ifdef MAIN

/* test driver  prints out either NMR_RAND values or a
   histogram        */

#include <stdio.h>

#define NMR_RAND        5000
#define MAX_BINS        500

#ifdef NO_PROTO
void main(argc, argv)
int argc;
char **argv;
#else
void main(int argc, char **argv)
#endif
{
    int j,k,nmr_bins,seed;
    int bins[MAX_BINS];
    double randm, bin_inc;
    double bin_limit[MAX_BINS];

    if ( argc != 3 )
    {
        printf("Usage -- %s nmr_bins seed\n", argv[0]);
        exit(1);
```

```
}

nmr_bins = atoi( argv[1] );
if ( nmr_bins > MAX_BINS )
{
    printf("ERROR -- maximum number of bins is %d\n",
            MAX_BINS);
    exit(1);
}

seed = atoi( argv[2] );

r250_init( seed );

if ( nmr_bins < 1 )     /* just print out the numbers */
{
    for (j = 0; j < NMR_RAND; j++)
        printf("%f\n", dr250() );
    exit(0);
}

bin_inc = 1.0 / nmr_bins;
for (j = 0; j < nmr_bins; j++)  /* init bins to zero */
{
    bins[j] = 0;
    bin_limit[j] = (j + 1) * bin_inc;
}

/* make sure all others are in last bin */
bin_limit[nmr_bins-1] = 1.0e7;

for (j = 0; j < NMR_RAND; j++)
{
    randm = r250() / (double)ALL_BITS;
    for (k = 0; k < nmr_bins; k++)
        if ( randm < bin_limit[k] )
        {
            bins[k]++;
            break;
        }
}
```

```
        for (j = 0; j < nmr_bins; j++)
            printf("%d\n", bins[j]);

}

#endif
```

H.3 Source Code for randlcg.h

```
/* randlcg.h     prototypes for the minimal standard random
                 number generator,

   Linear Congruential Method, the "minimal standard generator"
    Park & Miller, 1988, Comm of the ACM, 31(10), pp. 1192-1201


  rcsid: @(#)randlcg.h    1.1 15:48:09 11/21/94    EFC

*/

#ifndef _RANDLCG_H_
#define _RANDLCG_H_ 1.1

#ifdef __cplusplus
extern "C" {
#endif

#ifdef NO_PROTO
long          set_seed();
long          get_seed();
unsigned long int randlcg();

#else
long          set_seed(long);
long          get_seed(long);
unsigned long int randlcg();

#endif

#ifdef __cplusplus
}
#endif

#endif
```

H.4 Source Code for randlcg.c

```
/* rndlcg                Linear Congruential Method, the "minimal
                         standard generator"
                         Park & Miller, 1988, Comm of the ACM,
                        31(10), pp. 1192-1201
*/

static char rcsid[] = \
            "@(#)randlcg.c    1.1 15:48:15 11/21/94   EFC";

#include <math.h>
#include <limits.h>

#define ALL_BITS     0xffffffff

static long int quotient  = LONG_MAX / 16807L;
static long int remainder = LONG_MAX % 16807L;

static long int seed_val = 1L;

long set_seed(long int sd)
{
        return seed_val = sd;
}

long get_seed()
{
        return seed_val;
}


unsigned long int randlcg() /*returns a random unsigned int*/
{
if ( seed_val <= quotient )
        seed_val = (seed_val * 16807L) % LONG_MAX;
else
{
        long int high_part = seed_val / quotient;
        long int low_part  = seed_val % quotient;
```

```
long int test = 16807L * low_part \
   - remainder * high_part;

if ( test > 0 )
        seed_val = test;
else
                seed_val = test + LONG_MAX;

        }

        return seed_val;
}
```

Bibliography

Ackley, D. H. & Littman, M. L. (1994), Altruism in the evolution of communication, *in* Brooks & Maes (1994), pp. 40–48.

Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K. & Watson, J. D. (1989), *Molecular Biology of the Cell*, Garland Publishing Inc., New York. Second edition.

Ashiru, I. & Czarnecki, C. A. (1997), Evolving cooperative controllers for teams of robots, *in* C. A. Czarnecki, ed., 'Workshop on Recent Advances in Mobile Robots', De Montfort University, Leicester, pp. 58–66.

Axelrod, R. (1984), *The Evolution of Cooperation*, Basic Books, New York. Published in Penguin Books, 1990.

Bäck, T. (1992), Self-adaptation in genetic algorithms, *in* Varela & Bourgine (1992), pp. 263–271.

Brookes, M. (1998), 'Day of the mutators', *New Scientist* **157**(2121), 38–42.

Brooks, R. A. & Maes, P., eds (1994), *Artificial Life IV*, The MIT Press (A Bradford Book), Cambridge, MA.

Bryant, J. (1997), 'All the world's a stage: Using drama theory to resolve confrontations', *OR Insight* **10**(4), 14–21.

Cohen, D. & Eshel, I. (1976), 'On the founder effect and the evolution of altruistic traits', *Theoretical Population Biology* **10**, 276–302.

Cournot, A. A. (1838), *Recherches sur les Principes Mathematiques de la Theorie des Richesses*, Hachette, Paris. Published in English by Macmillan, New York as *Researches into the Mathematical Principles of the Theory of Wealth*, 1897, and reprinted by Augustus M. Kelley, New York, 1971.

Darwin, C. (1859), *The Origin of Species by Means of Natural Selection*, John Murray. Reprinted by Penguin, London, 1985.

Davis, L. (1991), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold. New York.

Dawkins, R. (1976), *The Selfish Gene*, Oxford. New edition, 1989.

Dawkins, R. (1979), 'Twelve misunderstandings of kin selection', *Zeitschrift für Tierpsychologie* **51**, 184–200.

Dawkins, R. (1983), *The Extended Phenotype*, Oxford.

Deb, K. & Beyer, H.-G. (2001), 'Self-adaptive genetic algorithms with simulated binary crossover', *Evolutionary Computation* **9**(2), 197–221.

Deb, K. & Goldberg, D. E. (1991), mGA in C: A messy genetic algorithm in C, Technical Report 91008, Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign.

Drake, J. W. (1991), 'A constant rate of spontaneous mutation in DNA-based microbes', *Proc. Natl. Acad. Sci. USA* **88**, 7160–7164.

Eldredge, N. & Gould, S. J. (1972), Punctuated equilibria: An alternative to phyletic gradualism, *in* T. J. M. Schopf, ed., 'Models in Paleobiology', Freeman Cooper, San Francisco, pp. 82–115.

Evans, D. (1998), 'The arbitrary ape', *New Scientist* **159**(2148), 32–35.

Fisher, R. A. (1930), *The Genetical Theory of Natural Selection*, Clarendon Press, Oxford.

Frean, M. R. & Abraham, E. R. (2001), 'A voter model of the spatial prisoner's dilemma', *IEEE Transactions on Evolutionary Computation* **5**(2), 117–121.

Gibbons, R. (1992), *A Primer in Game Theory*, Harvester Wheatsheaf, Hemel Hempstead.

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley.

Goldberg, D. E. & Richardson, J. (1987), Genetic algorithms with sharing for multimodal function optimization, *in* J. J. Grefenstette, ed., 'Proc. Second International Conference on Genetic Algorithms', Lawrence Erlbaum, pp. 41–49.

Gould, S. J. (1980), *The Panda's Thumb*, W. W. Norton & Co. Reprinted in Penguin Books, 1990.

Grafen, A. (1984), Natural selection, kin selection and group selection. *in* J. R. Krebs & N. B. Davies, eds, 'Behavioural Ecology: An Evolutionary Approach', Blackwell, Oxford, pp. 62–84. Second Edition.

Grefenstette, J. J. (1999), Evolvability in dynamic fitness landscapes: A genetic algorithm approach, *in* 'Proc. 1999 Congress on Evolutionary Computation (CEC99)', IEEE Press, Washington, DC, pp. 2031–2038.

Haddadi, A. (1996), *Communication and Cooperation in Agent Systems: A Pragmatic Theory*, Springer, Berlin.

Haldane, J. B. S. (1932), *The Causes of Evolution*, Longmans, Green, London.

Haldane, J. B. S. (1955), 'Population genetics', *New Biology* **18**, 34–51.

Hamilton, W. D. (1964), 'The genetical evolution of social behaviour (I and II)', *Journal of Theoretical Biology* **7**, 1–16; 17–52.

Hardin, G. (1968), 'The tragedy of the commons', *Science* **162**, 1243–1248.

Hillis, W. D. (1992), Co-evolving parasites improve simulated evolution as an optimization procedure, *in* Langton et al. (1992), pp. 313–324.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press. MIT Press edition, 1992.

Holzmüller, W. (1981), *Makromoleküle als Träger von Lebensprozessen*, Akademie-Verlag, Berlin. Published in English by Cambridge University Press as *Information in biological systems: the role of macromolecules*, 1984.

Howard, N. (1994), 'Drama theory and its relation to game theory (I and II)', *Group Decision and Negotiation* **3**, 187–206; 207–235.

Howard, N., Bennett, P., Bryant, J. & Bradley, M. (1993), 'Manifesto for a theory of drama and irrational choice', *Systems Practice* **6**, 429–434.

Jones, S. (1993), *The Language of the Genes*, HarperCollins. London. Published by Flamingo, with amendments and supplementary bibliographic essay, 1994.

Koza, J. R. (1992), *Genetic Programming*, MIT Press.

Koza, J. R. (1994), *Genetic Programming II*, MIT Press.

Krebs, J. R. & Davies, N. B., eds (1991), *Behavioural Ecology: An Evolutionary Approach*, Blackwell, Oxford. Third Edition.

Langton, C., ed. (1989), *Artificial Life*, Addison-Wesley.

Langton, C. G., ed. (1994), *Artificial Life III*, Addison-Wesley.

Langton, C., Taylor, C., Farmer, J. D. & Rasmussen, S., eds (1992), *Artificial Life II*, Addison-Wesley.

Lawrence, E., ed. (1995), *Henderson's Dictionary of Biological Terms*, 11 edn, Longman Scientific and Technical.

Li, J.-P., Balazs, M. E., Parks, G. T. & Clarkson, P. J. (2002), 'A species conserving genetic algorithm for multimodal function optimization', *Evolutionary Computation* **10**(3), 207–234.

Lomborg, B. (1996), 'Nucleus and shield: The evolution of social structures in the iterated prisoner's dilemma', *American Sociological Review* **61**, 278–307.

Matessi, C. & Jayakar, S. D. (1976), 'Conditions for the evolution of altruism under Darwinian selection', *Theoretical Population Biology* **9**, 360–387.

Maynard Smith, J. (1958), *The Theory of Evolution*, Penguin Books. Canto third edition published by Cambridge University Press, 1993.

Maynard Smith, J. (1982*a*), *Evolution and the Theory of Games*, Cambridge University Press.

Maynard Smith, J. (1982*b*), The evolution of social behaviour—a classification of models, *in* King's College Sociobiology Group, ed., 'Current Problems in Sociobiology', Cambridge University Press, pp. 29–44.

Maynard Smith, J. (1989), *Did Darwin Get It Right?*, Chapman and Hall. Published by Penguin, London, 1993.

Maynard Smith, J. & Szathmáry, E. (1995), *The Major Transitions in Evolution*, W. H. Freeman, Oxford.

Michalewicz, Z. (1994), *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag. Second, Extended Edition.

Mitchell, M. (1996), *An Introduction to Genetic Algorithms*, MIT Press.

Mor, Y., Goldman, C. V. & Rosenschein, J. S. (1996), Learn your opponent's strategy (in polynomial time)!, *in* G. Weiss & S. Sen, eds, 'Adaptation and Learning in Multi-Agent Systems', Springer, Berlin, pp. 164–176.

Murphy, M. P. & O'Neill, L. A. J., eds (1995), *What is Life? The Next Fifty Years*, Cambridge University Press.

Nasar, S. (1998), *A Beautiful Mind*, Faber and Faber, London.

Nash, J. F. (1950*a*), 'The bargaining problem', *Econometrica* **18**, 155–162.

Nash, J. F. (1950*b*), 'Equilibrium points in N-person games', *Proceedings of the National Academy of Sciences of the United States of America* **36**, 48–49.

Nash, J. F. (1951), 'Non-cooperative games', *Annals of Mathematics* **54**, 286–295.

Nash, J. F. (1953), 'Two person cooperative games', *Econometrica* **21**, 128–140.

Nijssen, S. & Bäck, T. (2003), 'An analysis of the behaviour of simplified evolutionary algorithms on trap functions', *IEEE Transactions on Evolutionary Computation* **7**(1), 11–22.

Nordin, P., Francone, F. & Banzhaf, W. (1996), Explicitly defined introns and destructive crossover in genetic programming, *in* P. J. Angeline & K. Kinnear, eds, 'Advances in Genetic Programming', MIT Press, Cambridge, MA, pp. 111–134.

Oliphant, M. (1994), Evolving cooperation in the non-iterated prisoner's dilemma: The importance of spatial organization, *in* Brooks & Maes (1994), pp. 349–352.

Potter, M. A. & De Jong, K. A. (1994), A cooperative coevolutionary approach to function optimization, *in* Y. Davidor, H.-P. Schwefel & R. Männer, eds, 'Parallel Problem Solving from Nature—PPSN III', Springer-Verlag, Berlin, pp. 249–257.

Randerson, J. (2003), 'Together we are stronger', *New Scientist* **177**(2386). 1–4. Special pull-out section.

Reeves, C. R., ed. (1995), *Modern Heuristic Techniques for Combinatorial Problems*, Advanced Topics in Computer Science Series, McGraw-Hill, London.

Reynolds, C. W. (1994), Competition, coevolution and the game of tag, *in* Brooks & Maes (1994), pp. 59–69.

Riolo, R. L., Cohen, M. D. & Axelrod, R. (2001), 'Evolution of cooperation without reciprocity', *Nature* **414**(6862), 441–443.

Riolo, R. L., Cohen, M. D. & Axelrod, R. (2002), 'Behavioural evolution (communication arising): Does similarity breed cooperation?', *Nature* **418**(6897), 500.

Roberts, G. & Sherratt, T. N. (2002), 'Behavioural evolution (communication arising): Does similarity breed cooperation?', *Nature* **418**(6897), 499–500.

Rose, S. (1991), *The Chemistry of Life*, Penguin, London. Third edition.

Rosenschein, J. S. & Zlotkin, G. (1994), *Rules of Encounter*, Artificial Intelligence Series, The MIT Press, Cambridge, MA.

Rudolph, G. (1997), 'Local convergence rates of simple evolutionary algorithms with cauchy mutations', *IEEE Transactions on Evolutionary Computation* **1**(4), 249–258.

Rudolph, G. (2001), 'Self-adaptive mutations may lead to premature convergence', *IEEE Transactions on Evolutionary Computation* **5**(4), 410–414.

Sapp, J. (1994), *Evolution by Association: A History of Symbiosis*, Oxford.

Sareni, B. & Krähenbühl, L. (1998), 'Fitness sharing and niching methods revisited', *IEEE Transactions on Evolutionary Computation* **2**(3), 97–106.

Schrödinger, E. (1944), *What is Life?*, Cambridge University Press. Canto edition with Autobiographical Sketches and Forward to *What is Life?* by Roger Penrose, 1992.

Schwefel, H.-P. (1995), *Evolution and Optimum Seeking*, John Wiley & Sons.

Sigmund, K. (1993), *Games of Life*, Oxford. Published in Penguin Books. 1995.

Spencer, H. (1864), *The Principles of Biology*, Vol. 1, Williams and Norgate, London and Edinburgh.

Stephens, C. R., Garcia Olmedo, I., Mora Vargas, J. & Waelbroeck, H. (1998), 'Self-adaptation in evolving systems', *Artificial Life* **4**(2), 183–201.

Taddei, F., Radman, M., Maynard-Smith, J., Toupance, B., Gouyon, P. H. & Godelle, B. (1997), 'Role of mutator alleles in adaptive evolution', *Nature* **387**, 700–702.

Trivers, R. L. (1971), 'The evolution of reciprocal altruism', *Quarterly Review of Biology* **46**, 35–57.

van Nimwegen, E. & Crutchfield, J. P. (1999), Metastable evolutionary dynamics: Crossing fitness barriers or escaping via neutral paths?, Technical Report 99-06-041, Santa Fe Institute. Santa Fe Institute Working Paper, submitted to *Bull. Math. Biol.*

Varela, F. J. & Bourgine, P., eds (1992), *Proc. First European Conference on Artificial Life*, MIT Press, Cambridge MA.

von Neumann, J. & Morgenstern, O. (1944), *Theory of Games and Economic Behaviour*, Princeton University Press.

Vose, M. D. (1999), *The Simple Genetic Algorithm Foundations and Theory*, MIT Press.

Watson, T. (1994), A new representation technique for genetic algorithms. *in* M. Keane, P. Cunningham, M. Brady & R. Byrne, eds, 'Proc. Seventh Irish Annual Conference of AI and Cognitive Science', Dublin University Press, pp. 233–246.

Watson, T. (1996), Kin selection and cooperating agents, Technical report. School of Computing Sciences, De Montfort University.

Watson, T. & Messer, P. (1999), Mutation genes in dynamic environments. *in* R. John & R. Birkenhead, eds, 'Soft Computing Techniques and Applications', Advances in Soft Computing, Physica-Verlag, Heidelberg. pp. 152–157.

Wilson, D. S. (1975), 'A theory of group selection', *Proceedings of the National Academy of Sciences of the U.S.A.* **72**, 143–146.

Wolpert, D. H. & Macready, W. G. (1997), 'No free lunch theorems for optimization', *IEEE Transactions on Evolutionary Computation* **1**(1). 67–82.

Wooldridge, M., Müller, J. & Tambe, M., eds (1996), *Intelligent Agents II: Agent Theories, Architectures, and Languages*, Springer, Berlin.

Wright, S. (1931), 'Evolution in Mendelian populations', *Genetics* **16**. 97–159.

Wright, S. (1945), 'Tempo and mode in evolution—a critical review', *Ecology* **26**, 415–419.

Wynne-Edwards, V. C. (1962), *Animal Dispersion in Relation to Social Behaviour*, Oliver & Boyd, Edinburgh.

Zermelo, E. (1913), Uber eine anwendung der mengenlehre auf die theorie des schachspiels, *in* E. W. Hobson & A. E. H. Love, eds, 'Proceedings of the Fifth International Congress of Mathematicians', Vol. 2, Cambridge University Press, pp. 501–504.